**MASTER EN SCIENCES INFORMATIQUES**

# Techniques d'implémentation du langage prolog.

Clantin, Marcel

*Award date:*
1985

*Awarding institution:*
Universite de Namur

Facultès Universitaires Notre-Dame de la Paix. Namur.

# TECHNIQUES D'IMPLEMENTATION DU LANGAGE PROLOG.

CLANTIN MARCEL.

Thèse présentée en vue de l'obtention du grade de licencié et maître en Informatique. Septembre 1985.

Sous la direction de Monsieur Henry Leroy.

Remerciements.

Je remercie en premier lieu Messieurs Axel van Lamsweerde et Henry Leroy, tous deux professeurs aux Facultès Notre-Dame de la Paix à Namur, le premier pour avoir, sans doute involontairement succitè ma curiositè lors de l'un de ses cours et le second pour m'avoir fourni l'occasion d'aborder un sujet qui ne manque pas d'intèrêt.

Je tiens à remercier tout particulièrement Monsieur Jean-François Perrot, professeur à l'Institut de Programmation de Paris VI, qui a bien voulu m'accueillir au sein de son èquipe et me consacrer un peu de son temps. Ses conseils sur un premier canevas de ce travail me furent prècieux. Je lui suis également reconnaissant de m'avoir servi de guide dans le dèpouillement de l'abondante littèrature sur le sujet.

Ma reconnaissance va également à Messieurs Patrice Boizumault et Marc Tizzano pour l'aide dèsintèressèe qu'ils m'ont apportèe lors de mes premiers contacts avec l'interprèteur Prolog.

Je citerai principalement à l'ordre du jour Monsieur Yves Deville qui a patiemment relu cet ouvrage et dont la critique fut des plus constructives.

Enfin, je remercie toutes les personnes qui de près ou de loin ont contribuè à l'èlaboration de cette thèse.

# Table des matières.

# Introduction.

Prolog est un langage de programmation simple mais puissant basé sur la logique symbolique. Le présent travail a pour objet d'analyser la philosophie d'implémentation de celui-ci. Cependant, il nous semble au préalable utile de resituer les origines de ce nouveau langage ainsi que d'évoquer brièvement quelques applications où il peut trouver emploi.

## 1. Un courant de pensée.

Les divers systèmes Prolog découlent de toute une école de pensée ayant à sa base le calcul des prédicats créé il y a plus d'un siècle par Gottlob Frege. Au long des ans, ce système original fut amélioré par des logiciens tels Bertrand Russell, David Hilbert, Kurt Gödel, Alonzo Church, Alan Turing et surtout Jacques Herbrand dont les travaux inspirèrent largement les premiers démonstrateurs automatiques.

Les premiers essais butèrent de suite contre le grave problème de la complexité de calcul. Des recherches visant à sa résolution furent donc entreprises et menèrent au développement de nouveaux systèmes logiques, équivalents aux systèmes traditionnels mais nettement mieux adaptés à une démonstration automatique efficace. Le plus remarquable de ceux-ci est certainement le principe de résolution de John Adam Robinson. Cordell Green proposa également l'emploi de systèmes de résolution lors de l'implémentation de "Query Systems" et finalement Robert Kowalsky en vint à développer l'interprétation procédurale de la logique qui sera à l'origine de l'emploi de celle-ci en tant que langage de programmation.

La mise en oeuvre d'un tel projet nécessita un travail en deux temps :

- d'abord, il fallut mettre au point des systèmes logiques utilisables pour l'expression de problèmes grandeur nature;
- ensuite on passa à la construction de démonstrateurs adaptés à ces logiques et qui purent être utilisés d'une manière pratique en tant que machine logique.

Aujourd'hui de tels systèmes sont opérationnels et les plus développés sont les interpréteurs de la famille Prolog. Ceux-ci proviennent principalement des travaux de deux groupes de recherche :

- le groupe d'Intelligence Artificielle (G.I.A.) de Marseille-Luminy dont les travaux notamment d'Alain Colmerauer et de Philippe Roussel dans le cadre de recherches sur la compréhension du langage naturel amenèrent la mise sur pied du premier

interpréteur. Son aire d'influence est principalement la France.
- l'Imperial College de Londres et l'université d'Edimburgh en Ecosse avec les travaux de David Warren dont l'impact porte plutôt sur la zone de culture anglo-saxonne.

De nombreuses autres implémentations existent. Il serait fastidieux de toutes les recenser mais citons cependant la version hongroise, [SZER, 1977], celle de l'université de Waterloo, Canada, [ROBE, 1977] ou encore les travaux de Maurice Bruynooghe à la K.U.L. [BRUY, 1976].

## 2. Champ d'application du langage.

Le langage Prolog trouve utilisation dans une multitude de domaines. La motivation première de sa création fut le traitement du langage naturel. Cependant, Prolog s'adapte assez bien à certaines applications en Intelligence Artificielle et son emploi pour la réalisation de bases de données procure de nombreux avantages relativement aux systèmes relationnels. Notons également que Prolog n'est pas inadapté au traitement classique des données, bien que dans ce domaine, le programmeur ressente rapidemment les contraintes imposées par la faiblesse du langage en matière de fichiers.

L'engouement pour le langage est en fait relativement récent et date de l'annonce du "Projet japonais d'ordinateurs de cinquième génération" au tout début des années 1980. Ce projet vise à développer des ordinateurs à très hautes performances, impliquant en phase ultime l'utilisation d'architectures parallèles. Prolog se trouva être choisi pour en constituer le langage de base principalement en fonction de ses nombreuses ouvertures au parallélisme. Actuellement, ce projet a débouché sur la mise au point de deux machines, PSIM : Personal Sequential Inference Machine et PIM : Parallel Inference Machine, mais le langage a été fortement modifié pour y aboutir. Le résultat pratique de ces recherches laisse d'ailleurs encore une grande partie de la communauté scientifique dans le scepticisme.

## 3. Plan du travail.

Dans le présent travail, nous nous proposons d'étudier les techniques utilisées lors du développement d'un interpréteur du langage Prolog. Plutôt que de décortiquer un par un les interpréteurs réalisés par les auteurs précités, nous avons préféré tenter de dégager les grandes lignes de la conception d'un système Prolog, relever les problèmes rencontrés au cours de son développement ainsi qu'analyser et comparer les solutions apportées à ceux-ci par différents concepteurs.

Dans ce but, nous avons organisé notre ouvrage autour de trois grands axes.

Le premier, constitué des chapitres un et deux, resitue le langage Prolog dans le contexte théorique qui est à l'origine de sa création. Le chapitre un rappelle d'abord la définition du sous-ensemble de la logique sur lequel s'appuie le langage tout en en arrêtant une syntaxe. Ensuite, nous y relèverons la richesse sémantique dont bénéficie ce langage. Enfin, les limitations de son formalisme seront rapidemment évoquées. Le chapitre deux nous rappelera les mécanismes de base découverts par John Robinson, à savoir l'unification et la résolution ainsi que les adaptations qui y furent apportées dans le cadre de l'interpréteur Prolog.

La deuxième partie constitue le coeur de notre étude. Elle s'organise en trois chapitres. Le chapitre trois, de loin le plus important, exposera les grandes lignes des structures de données nécessaires à l'interpréteur, à savoir le code source d'un programme et les piles de travail. Nous montrerons que l'organisation de ces dernières est le fruit d'une mûre réflexion et se révèle être la plus efficace. Le chapitre quatre, assez bref, étudiera les modes de représentation des variables. Le sujet est relativement important car les choix posés à ce niveau sont très déterminants pour la construction d'un interpréteur. Enfin, le chapitre cinq portera sur un point non négligeable, à savoir les méthodes mises en oeuvre pour réduire la consommation d'espace de travail. Il s'agit de diverses techniques de récupération de zones de mémorisation devenues sans intérêt. Nous verrons que leur mise en place aboutit à un profil très semblable de la zone de travail quelque soient les choix de conception posés.

Le dernier axe reprendra avec le chapitre six un ensemble représentatif d'outils fournis au programmeur par tout système Prolog. Essentiellement, il s'agira des opérations d'entrée et de sortie, de moyens de contrôle de l'interpréteur et des opérations arithmétiques. Pour terminer, le chapitre sept recensera diverses voies d'amélioration du système. A ce sujet, nous noterons la nécessité de compléter l'environnement de programmation, nous étudierons en détails la réalisation de David Warren ainsi que les ouvertures que présente Prolog au parallélisme et nous conclurons par un relevé de la recherche en matière de retour arrière intelligent.

Le lecteur interressé trouvera enfin en annexe le développement rapide et incrémental d'un petit interpréteur écrit en Vlisp.

En utilisant une importante masse de publications sur le sujet, nous espérons apporter à celles-ci une unité qui leur fait généralement défaut. Nous osons croire que le lecteur trouvera ici un outil utile à sa compréhension des mécanismes profonds de l'interpréteur du langage Prolog. En outre, nous espérons également que ce travail pourra servir aux analystes et programmeurs désireux de développer rapidement un émulateur de Prolog qui lui permettra de tester le langage avant l'acquisition d'un système commercialisé.

# Chapitre 1: La representation des connaissances.

La representation des connaissances dans un systeme Prolog utilise un sous-ensemble de la logique clausale, en l'occurence les clauses de Horn. Dans ce chapitre, nous nous proposons d'en donner brievement une syntaxe et la definition pour ensuite en aborder la semantique. Nous evoquerons pour terminer quelques limitations induites par une telle formulation.

## 1. La logique clausale.

### 1.1. Une syntaxe generale.

Les objets syntaxiques de la logique clausale sont d'une part les connecteurs "NON", "OU", "ET" et "SI" que nous noterons respectivement "¯", "v", "·" et ":-" et d'autre part les litteraux.

Tout litteral note de maniere prefixee sera de la forme :

pr (t1 , t2 , ... , tn) avec n >= 0

où le symbole de tete, "pr" est appele predicat et où chacun des "ti" (i = 1..n) est un terme.

Il existe deux categories de termes :

- les termes simples, formés d'un seul symbole, variable ou constante. Une variable est caracterisee par un identificateur et un mecanisme de reconnaissance. En Prolog, l'utilisation d'une majuscule ou d'un caractère privilegié, generalement placé en tête de l'identificateur sera preferee à la technique des declarations prealables. Une constante est quant à elle ou bien une valeur numerique ou bien un atome, c'est à dire une chaine quelconque de caracteres.

Nota bene.

Dans la suite de ce travail, nous utiliserons toujours des lettres capitales pour composer les identificateurs des variables, reservant ainsi l'emploi des minuscules et des caracteres numeriques pour les constantes.

- les termes composés, de la forme :

$$f\ (t1\ ,\ t2\ ,\ \ldots\ ,\ tk) \quad \text{avec} \quad k > 0$$

où "f" est un symbole fonctionnel d'arité k et ses arguments, à savoir les "tj" (j = 1..k) sont eux-mêmes des termes. Nous utiliserons une notation préfixée pour la simple raison que celle-ci est de pratique plus habituelle en logique symbolique. Cependant, dans les cas où cela n'introduit aucune ambiguité, il serait tout aussi valable d'utiliser une notation infixée.

Remarques.

1. Un littéral sera parfois appelé terme booléen car il est lui-même un terme composé dont le symbole fonctionnel est le prédicat.

2. Les constantes peuvent être considérées comme des symboles fonctionnels d'arité nulle.

## 1.2. Identité de deux termes.

Deux termes simples sont identiques si et seulement s'ils sont formés du même symbole : mêmes identificateurs pour les variables, mêmes valeurs numériques ou mêmes atomes pour les constantes.

Deux termes composés sont identiques si et seulement s'ils possèdent même symbole fonctionnel et si leurs arguments sont respectivement identiques deux à deux.

Soient deux termes composés :

$$T1 = sf1\ (t11\ ,\ t12\ ,\ \ldots\ ,\ t1k) \quad \text{avec } k > 0,$$
$$T2 = sf2\ (t21\ ,\ t22\ ,\ \ldots\ ,\ t2j) \quad \text{avec } j > 0.$$

T1 est identique à T2 si et seulement si

- sf1 est identique à sf2,
- k = j,
- pour tout i = 1..k, t1i est identique à t2i.

## 1.3. La notion de clause.

Une clause est une expression logique du type :

$$C1 \vee C2 \vee \ldots \vee Cm \quad :- \quad A1 \cdot A2 \cdot \ldots \cdot An \qquad (1)$$
$$\text{avec } n,m >= 0.$$

où les "Ci" et les "Aj", (i = 1..m, j = 1..n) sont des littéraux.

La disjonction :

$$C1 \vee C2 \vee \ldots \vee Cm \qquad \text{avec } m >= 0$$

sera connue sous le nom de tête de la clause; les littéraux la composant portent le nom de conclusions ou encore conséquents de cette clause. La conjonction :

$$A1 \cdot A2 \cdot \ldots \cdot An \qquad \text{avec } n >= 0$$

formera, quant à elle, son corps. Ces littéraux sont les conditions ou antécédents de la clause, plus souvent désignés sous le nom de prémisses.

Une formulation équivalente de (1) est

$$C1 \vee C2 \vee \ldots \vee Cm \vee \neg A1 \vee \neg A2 \vee \ldots \vee An \qquad (2)$$
$$\text{avec } n,m >= 0.$$

Nous en concluons que toute clause est une disjonction de littéraux négatifs et positifs.

La forme clausale de la logique a la même puissance d'expression que le calcul des prédicats. Il est donc toujours possible de transformer une formule du calcul des prédicats en une forme clausale. [KOWA, 1979b] et [CLOC et alt., 1984]. En effet, toutes les variables apparaissant au sein d'une clause sont implicitement gouvernées par le quantificateur universel "pour tout". L'emploi du quantificateur existentiel est quant à lui évité grâce à un artifice consistant à utiliser des constantes ou des symboles fonctionnels pour désigner certains individus (fonction de Skolem).

## 1.4. Les clauses de Horn.

Une clause de Horn est une clause qui possède au plus un seul littéral positif. Autrement dit, une clause du type (1) ou (2) est une clause de Horn si et seulement si m est inférieur ou égal à un.

Robert Kowalski [KOWA 1979a et b] classe ces clauses selon quatre catégories :

1. (m = 1 et n > 0).
   C :- A1 ^ A2 ^ ... ^ An avec n > 0.
   Règle ou axiome.

2. (m = 1 et n = 0).
   C :- .
   Fait ou assertion.

3. (m = 0 et n > 0).
   :- A1 ^ A2 ^ ... ^ An avec n > 0.
   Démenti.

4. (m = 0 et n = 0).
   :- .
   La clause vide.

## 2. Sémantique du langage Prolog.

Une des principales caractéristiques de ce langage est qu'il bénéficie d'une triple sémantique. L'interprétation déclarative des clauses de Horn découle d'une manière fort naturelle de la logique. Robert Kowalski nous en propose une deuxième, procédurale. La troisième interprétation est celle de la réduction des problèmes, généralement utilisée dans la génération de plans en robotique ou pour les réponses aux interrogations de bases de données. Kowalski la prétend, avec raison, identique à la deuxième.

### 2.1. L'interprétation déclarative.

D'une manière générale, nous pouvons dégager l'interprétation déclarative d'une règle que celle-ci contienne ou non des variables (soient X1, X2, ... , Xk avec k >= 0).

Quelles que soient les valeurs de X1, X2, ... ,
Xk (k >= 0),
si (A1 ^ A2 ^ ... ^ An) est vrai,
alors (C) est vrai.

soit, exprimé d'une autre manière, que le littéral positif est une conséquence logique de la conjonction des antécédents.

Dès lors, si n = 0, il s'agira d'une affirmation inconditionnelle d'où le nom d'assertion :

> Quelles que soient les valeurs de X1, X2, ... ,
> Xk (k >= 0),
> (C) est vrai.

Pareillement, un démenti aura la signification :

> Quelles que soient les valeurs de X1, X2, ... ,
> Xk (k >= 0),
> (A1 ^ A2 ^ ... ^ An) est faux.

ou ce qui revient au même :

> Pour aucune des valeurs que puissent prendre X1,
> X2, ... , Xk (k >= 0),
> (A1 ^ A2 ^ ... ^ An) ne sera vrai.

Par convention, la clause vide s'interprète toujours en faux.

Un ensemble d'axiomes et d'assertions constitue un programme pour le système. Ainsi que le résume David Warren [WARR et alt., 1977], [WARR, 1977], la sémantique déclarative du langage Prolog définit l'ensemble des prédicats dont on peut affirmer qu'ils sont vrais en fonction du programme.

Dans la pratique, pour comprendre une clause déclarativement, il suffit d'interpréter celle-ci comme l'abréviation d'une phrase du langage naturel. Chaque symbole fonctionnel, et donc chaque littéral, ne porte aucune autre signification intrinsèque que celle que le programmeur veut bien lui donner. Celle-ci sera unique.

Ainsi, si nous considérons par exemple l'ensemble de clauses suivant :

```
concatenate (nil , LA , LA) :- .
concatenate (cons (X , LA) , LB , cons (X , LC)) :-
                                    concatenate (LA , LB , LC).

reverse (nil , nil) :- .
reverse (cons (X , LA) , LB) :-
 reverse (LA , LC) ^ concatenate (LC , cons (X , nil) , LB).
```

définissant la concaténation et le renversement de listes, nous pouvons appliquer à chaque symbole fonctionnel une interprétation, à savoir :

- les symboles X, LA, LB, LC sont des variables. Chaque variable doit être interprétée comme un objet arbitraire.
- "nil" désigne la liste vide.
- "cons" est un constructeur de liste; ainsi cons (X , LA) signifie la liste dont le premier élément est X et les suivants la liste LA.
- "concatenate (LA , LB , LC)" signifie la concaténation de la liste LA avec la liste LB donne la liste LC.
- "reverse (LA , LB)" veut dire le renversement de la liste LA donne la liste LB.

Il va de soi qu'un programmeur avisé devrait toujours se contraindre à définir de manière explicite chacun des symboles fonctionnels et des prédicats de son programme.

Notons encore que la portée de chaque variable est limitée à la clause où elle apparaît, de sorte que des variables de même nom apparaissant dans des clauses différentes doivent être considérées comme n'ayant aucun rapport entre elles.

Par conséquent, notre programme peut s'interpréter comme l'abréviation des quatre phrases suivantes :

- La concaténation de la liste vide avec la liste LA est la liste LA.

- La liste dont le premier élément est X et les autres LA concaténée avec la liste LB donne la liste dont le premier élément est X et les autres LC si la concaténation de la liste LA avec la liste LB donne la liste LC.

- Le renversement d'une liste vide est une liste vide.

- Le renversement de la liste dont le premier élément est X et les autres LA est la liste LB si le renversement de la liste LA est la liste LC et si la concaténation de LC avec la liste dont l'unique élément est X donne la liste LB.

## 2.2. L'interprétation procédurale.

L'interprétation déclarative du langage ne fait aucune référence à l'ordre des clauses dans le programme ou des antécédents dans une clause. Cet ordre représente de l'information de contrôle et relève de l'interprétation procédurale des clauses de Horn. Celle-ci ressort d'un point de vue radicalement différent dû à Robert Kowalski. [EMDE et alt., 1976].

Ainsi la clause :

C :- A1 ^ A2 ^ ... ^ An avec n >= 0

que nous pouvons encore ecrire :

pr (t1 , t2 , ... , tk) :- A1 ^ A2 ^ ... ^ An (3)
avec k > 0,n >= 0,

sera traitee en tant que declaration de la procedure de nom "pr". Le consequent sera considere comme un point d'entree dans cette procedure tandis que les antecedents formeront son corps. Chaque litteral negatif "Ai" (i = 1..n) sera lui-meme interprete comme un appel de procedure.

Des lors, une clause comme (3) s'interpretera de la sorte : pour executer la procedure "pr" sur les arguments (t1, t2, ..., tk) avec k > 0, il suffit d'executer successivement les appels A1, A2, ..., An avec n >= 0. Le cas particulier d'une assertion sera considere comme une definition de procedure avec un corps vide dont l'execution se termine immediatement.

La definition complete d'une procedure ou paquet est un ensemble non vide de clauses de type axiome ou assertion ayant toutes le meme predicat dans leur tete. Une procedure Prolog possede de la sorte plus d'un point d'entree. Un ensemble de definitons de procedures constitue un programme.

L'interpretation procedurale met en evidence la façon dont un objectif donne sera prouve. L'execution de cette preuve sera declenchee par un dementi dont chaque litteral sera vu comme une invocation de procedure. La clause vide sera quant a elle interpretee comme une condition d'arret.

## 2.3. Lien entre les deux semantiques.

Les deux semantiques presentees ci-dessus possedent une difference fondamentale. L'interpretation declarative est d'ordre definitionnelle. Par consequent, l'ordre des clauses d'un programme ou des litteraux dans le corps d'une clause est sans aucune espece d'importance. La semantique procedurale est d'ordre algorithmique. La sequence des appels dans chaque clause ou des clauses au sein de la definition d'une procedure garantit le resultat correct du programme.

Les interpreteurs Prolog implementent comme nous l'avons dit les deux interpretations permettant en principe la lecture des programmes sous les deux points de vue. Cependant en pratique, la semantique declarative se perd rapidemment lorsque le programmeur fait usage de predicats evaluables a effet de bord dont l'usage vise d'ailleurs souvent a faciliter la programmation procedurale.

## 3. Limitations du formalisme.

Le point de vue central de la programmation en logique est de décomposer tout algorithme en une partie logique et une partie de contrôle. Pour une grande partie des problèmes courants de programmation, la partie logique peut être spécifiée sous forme de clauses de Horn. Cependant une critique généralement adressée à cette logique, et par voie de conséquence à un langage de programmation tel Prolog, est son aspect restrictif. Qu'en est-il réellement ? Dans cette section, nous nous proposons de relever les limitations inhérentes à cette forme particulière de la logique mais aussi de noter celles dues à l'utilisation du formalisme logique en soi.

### 3.1. Limitations inhérentes aux clauses de Horn.

Nous avons fait remarquer précédemment que l'utilisation de la logique clausale n'induisait aucune perte de puissance d'expression par rapport au calcul des prédicats.

Cependant, les clauses de Horn portent dans leur définition une restriction certaine par rapport à la logique clausale pure : jamais il n'est possible d'exprimer une disjonction dans la conclusion.

$$A \vee B :- C$$

De même, en général, il n'est guère aisé de traiter les définitions par équivalence (si et seulement si), parce que la partie "seulement si" nécessite souvent l'emploi d'une disjonction dans la conclusion. [GALL, 1981].

Cette restriction ne semble toutefois guère gênante : la majorité des problèmes classiques se formaliserait aisément à l'aide de clauses de Horn. [ROY , 1984]. Pour notre part, nous bornerons à signaler ces inconvénients mineurs. Le lecteur désireux d'examiner un point de vue plus théorique consultera le rapport de S. A. Tarnlund. [TARN, 1975].

Il est cependant généralement admis que la logique clausale n'est pas d'un emploi aussi naturel que la logique classique. Ceci peut se comprendre aisément sur un exemple. Comparons les deux clauses suivantes :

```
ancêtre (X , Y) :- parent (X , Y).
ancêtre (X , Y) :- parent (X , Z) ' ancêtre (Z , Y).
```

définissant :

- X est l'ancêtre de Y si X est un parent (père ou mère) de Y.
- X est l'ancêtre de Y si X est un parent de Z et si Z est l'ancêtre de Y.

à leur formulation quantifiée :

Pour tout X, Y, ancêtre (X , Y) :- parent (X , Y)
v ( Il existe Z | parent (X , Z) ^ ancêtre (Z , Y)).

définissant :

- Pour toute paire d'individus {X ,Y}, X est l'ancêtre de Y si X est un parent de Y ou s'il existe un autre individu Z tel que X soit un parent de Z qui lui-même serait l'ancêtre de Y.

La seconde formulation est évidemment plus parlante que la première. C'est la raison pour laquelle certains auteurs proposent d'exprimer le problème sous cette forme plus naturelle pour ensuite en dériver la forme clausale grâce à des transformations standards. [CLAR et alt., 1977] et [NILS, 1971].

## 3.2. Limitations du formalisme logique.

Nous pouvons admettre que le développement d'un programme correct et efficace peut se décomposer en deux sous-problèmes plus simples :

- La première tâche est (ou devrait être) la rédaction de spécifications correctes du problème à résoudre.
- Ensuite suivra une phase de construction et d'optimisation du programme.

Arguant que la langue naturelle est source d'imprécisions, d'ambiguités voire de contradictions, des auteurs comme Bertrand Meyer [MEYE, 1975] et Barbara Liskov [LISK et alt., 1975] proposent l'utilisation de langages formels de spécifications. Reprenant leur point de vue et partant du fait qu'un tel langage se doit également d'être naturel et simple d'emploi, Robert Kowalski [KOWA, 1982] prône l'utilisation de la logique symbolique. En outre, celui-ci fait astucieusement remarquer que ce langage est, sous une certaine forme, directement compréhensible par la machine, ce qui permet aisément de tester, corriger et éventuellement compléter les spécifications avant de passer en phase de codage.

De plus, et toujours selon Kowalski, dans le cas particulier de la programmation en logique, le langage de spécification sert aussi au codage, d'où simplification supplémentaire. En effet, tous les problèmes de traduction des spécifications en programme sont ainsi évités. En outre dans une majorité de cas, le

programme obtenu tournera avec une efficacité suffisante. Si tel n'est pas le cas, le programmeur pourra transformer soit la structure de contrôle du programme, soit sa composante logique, à l'aide de quelques outils fournis par le langage.

Cependant, cette position n'est pas unanime. Baudouin Lecharlier [LECH, 1985] relève en effet, qu'étant donné l'objectif de précision absolue qui préside à l'élaboration des formalismes, ceux-ci se révèlent tout à fait inadéquats en tant que langages de communication. Une spécification formelle sera soumise à toutes les limitations d'un énoncé formel. En ce qui concerne la logique du premier ordre, nous en donnerons pour preuve la nécessité d'introduire un ensemble de prédicats évaluables par exemple pour le calcul arithmétique et les opérations d'entrée et de sortie. Poursuivant par ailleurs son raisonnement, Baudouin Lecharlier en arrive à conclure que dans la pratique, un langage formel sera toujours aussi inadéquat qu'un langage de programmation pour communiquer à quoi sert le programme. Il n'en demeure pas moins que l'addition de spécifications formelles à la documentation conventionnelle d'un programme permet une vue stéréoscopique du problème. L'utilisation des deux méthodes procure ainsi la possibilité d'une vérification croisée. Ceci est seulement possible du fait que les deux notations sont tellement différentes que l'une ne peut être vue comme la traduction de l'autre. Les deux notations ne sont équivalentes qu'à un très haut niveau d'abstraction et doivent être dérivées de la solution conceptuelle indépendamment l'une de l'autre, ce qui diminue fortement la possibilité qu'elles contiennent la même erreur.

D'autre part, il est bon de remarquer que, très souvent, les adeptes de la programmation en logique manquent un peu d'esprit critique quant à sa puissance et ses limitations. Une des raisons de cet état de chose est certainement que ce type de programmation procure à son utilisateur des notations beaucoup plus proches de son mode de pensée que les langages classiques. Tout d'abord, le programme est dégagé des initialisations et assignations, ce qui ne peut que contribuer à sa clarté et à sa concision. Ensuite, et surtout, il y a la possibilité d'utilisation de la logique. Mais sur ce point, peut-être est-il bon de ne pas s'emballer. En effet, il n'est certainement pas plus aisé de développer des spécifications formelles correctes, même compréhensibles par la machine, que de produire directement du code conforme à une certaine spécification. La preuve en est l'importance qu'accorde Robert Kowalski à la possibilité de tester celles-ci de manière interactive : n'est-ce pas déjà là une forme de codage ?

Enfin, et en guise de conclusion à un débat qui sort du cadre de ce travail, nous noterons que quelques transformations de la composante logique ne suffiront généralement pas pour supprimer les inefficacités d'un programme; très souvent il pourra également être utile de pouvoir modifier la composante de contrôle, ce qui dans l'état actuel des choses est rarement possible et en tout cas n'est guère aisé.

# Chapitre 2: Mécanismes de base d'un système Prolog.

## 1. Utilisation habituelle d'un système Prolog.

La pratique habituelle de Prolog consiste à poser des questions au système dans le cadre de l'univers défini par le programme. Une question (Q1, Q2, ..., Qp) ? (avec p > 0 et chaque Qi (i = 1..p) un littéral) signifie

> "Est-ce que (Q1 ^ Q2 ^ ... ^ Qp) est vrai dans le cadre de la base de connaissances ?"

ou encore s'il y figure des variables comme ce sera très probablement le cas :

> "Existe-il des valeurs pour les variables X1, X2, ..., Xq (q >= 0) telles que (Q1 ^ Q2 ^ ... ^ Qp) soit vrai dans le cadre de la base de connaissances ?"

La démonstration de telles relations se fait par la mise en oeuvre d'un procédé particulier de résolution dans le cadre du calcul des prédicats où l'unification permet de sélectionner les clauses candidates à une déduction donnée. Le procédé général de résolution ainsi que l'algorithme d'unification furent introduits en 1965 par John Adam Robinson [ROBI, 1965] et depuis de nombreuses versions dérivées en ont été proposées par divers auteurs.

## 2. L'unification.

### 2.1. Définitions préliminaires.

#### 2.1.1. Substitution.

Une substitution S est un ensemble fini de couples {(t1,X1) , (t2,X2) , ... , (tn,Xn)} avec n >= 0 où

- les "ti" (i = 1..n) sont des termes,
- les "Xi" (i = 1..n) sont des variables,
- pour tout k = 1..n, Xk est distincte de tk,
- pour tous k,j = 1..n et k <> j, Xk est distincte de Xj.

Chaque variable à laquelle est associé un terme dans une substitution est dite liée à ce terme. Inversement, une variable non liée à un terme est libre.

Une substitution S peut être appliquée à un terme t, ce que nous noterons (t)S et lirons l'instanciation de t dans S, pour produire un nouveau terme t', obtenu en remplacant simultanément dans t chaque variable par le terme qui lui est lié dans la substitution S. Toute variable libre est recopiée littéralement.

Exemple :

Soient :

- X, Y des variables, f, g, h des symboles fonctionnels, a, b des constantes;
- un terme t : f (X , g(Y) , a);
- une substitution S : { (h(Y),X) , (b,Y) };

alors,

- (t)S = f (h(Y) , g(b) , a)

et

- ((t)S)S = f (h(b) , g(b) , a).

### 2.1.2. Unificateur de deux termes.

Un unificateur de deux termes t1 et t2 est une substitution S telle que les deux termes (t1)S et (t2)S soient identiques. Lorsqu'une telle subsitution existe, nous dirons que les deux termes t1 et t2 sont unifiables.

Exemples :

Les deux termes :

- t1 = f (X , h(X) , Y)
- t2 = f (g(Z) , W , Z)

sont unifiables car il existe une substitution

- S = { (g(Z),X) , (Z,Y) , (h(g(Z)),W) }

telle que (t1)S et (t2)S soient identiques. En effet,

- (t1)S = f (g(Z) , h(g(Z)) , Z)

- (t2)S = f (g(Z) , h(g(Z)) , Z)

Par contre, les termes :

- t3 = (f a b)
- t4 = (f X X)

ne le sont pas car il n'est pas possible de trouver une substitution S permettant d'affirmer que (t3)S et (t4)S sont identiques.

### 2.1.3. Unificateur le plus général.

Un unificateur U de deux termes t1 et t2 est dit le plus général si et seulement si pour tout autre unificateur U' de ces deux termes, il existe une substitution S telle que pour tout terme t, (t)U' = ((t)U)S. Autrement dit l'unificateur U' résulte de la composition, au sens où Robinson la définit [ROBI, 1965], de l'unificateur U et de la substitution S : U' = (S o U).

Lorsque deux termes sont unifiables, l'unificateur le plus général existe toujours. La preuve de cette affirmation se trouve dans la référence précitée.

## 2.2. Calcul de l'unificateur le plus général.

C'est là le coeur de tout système ayant à sa base le principe de résolution. A ce titre, sa performance est de la plus haute importance quant à l'efficacité de l'ensemble de l'interpréteur. Or, l'algorithme tel qu'il fut initialement proposé par John Robinson peut se révéler extrêmement inefficace et c'est la raison pour laquelle bien des auteurs ont cherché des moyens de l'améliorer. Nous ne citerons que [MART et alt., 1982] où le lecteur trouvera détails et références complémentaires sur ces investigations. Pour notre part, visant essentiellement la compréhension des principes qui sous-tendent son développement, nous reprendrons uniquement l'algorithme original de Robinson, bien qu'il soit très théorique. Nous éviterons pour l'instant de nous attarder aux détails pratiques de son implémentation pour la simple raison qu'ils sont étroitement liés aux structures de représentation des termes en mémoire, ce qui sera examiné dans les chapitres ultérieurs.

## 2.2.1. Principes sous-jacents à l'algorithme de Robinson.

L'idée centrale de l'algorithme de Robinson pour calculer l'unificateur le plus général d'un ensemble de termes (en incluant au moins deux) est d'établir successivement des substitutions de taille un (c'est à dire du type { (t,X) } dont chacune aura pour but de rendre identiques deux sous-expressions de deux termes différents et pour ensuite les composer et fournir le résultat.

Le problème revient donc principalement à identifier deux sous-expressions à unifier, à s'assurer que la substitution qui les unifie existe bien et, si tel est le cas à la construire. Chaque nouvelle substitution ainsi constituée est alors composée avec les autres pour finalement former l'unificateur cherché.

Il existe cependant des cas où cet unificateur ne pourra être obtenu : l'algorithme s'arrêtera alors en signalant l'échec. Quels sont ces cas ? Trivialement deux constantes différentes, deux termes composés de symboles fonctionnels non identiques ou d'arité différente ou encore une constante et un terme composé ne s'unifient pas.

Il existe cependant une autre possibilité d'échec : lorsque une variable apparaît dans le terme composé qui doit lui être lié. En effet dans ce cas, aucune substitution S ne pourra rendre une variable X et un terme t identiques, de part le fait que (X)S devient un sous-terme de (t)S. Par conséquent, X et t ne peuvent s'unifier. La vérification de l'existence d'une telle situation est connue sous le nom de "test d'occurence".

Exemple.

La substitution S = { (a,X) , (g(Y),Y) } n'est pas un unificateur des termes :

- t1 = f (X , Y)
- t2 = f (a , g(Y))

En effet, (Y)S = g(Y) est un sous-terme de (g(Y))S = g(g(Y)) avec pour conséquence que les termes :

- (t1)S = f (a , g(Y))
- (t2)S = f (a , g(g(Y)))

ne sont pas identiques.

### 2.2.2. L'algorithme d'unification de Robinson.

Nous reprenons ici l'algorithme tel que John Robinson le propose régulièrement, [ROBI, 1965], [ROBI, 1968], [ROBI, 1971], mais sous une présentation un peu plus commentée afin de permettre au lecteur de se resituer par rapport aux idées maîtresses mises en évidence précédemment.

Soient { T1, T2, ... , Tn } (avec n > 1) l'ensemble des termes à unifier :

1. (* Initialisations. *)
   S := {} (* la substitution vide *) ;
2. (* Test de terminaison. *)
   Si (T1)S == (T2)S == ... == (Tn)S
   - alors : Stop ;
   - sinon : passer à l'étape 3. ;
3. (* Localisation de deux sous-termes à unifier. *)
   Soit k, le plus petit nombre inférieur ou égal à n tel que (Tk)S diffère de (T[k-1])S ;
   E := (Tk)S ; F := (T[k-1])S ;
   Analyser E et F parallèlement et de gauche à droite pour localiser la position la plus à gauche telle que les termes E et F diffèrent ;
   Soient E' et F', les sous-expressions de respectivement E et F qui commencent à cette position ;
4. (* Les deux sous-expressions sont-elles unifiables ? *)
   Si ni E' ni F' ne sont des variables
   - alors : pas d'unificateur ; Stop ;
   - sinon : passer à l'étape 5 ;
5. (* Test d'occurence *)
   Si soit E', soit F' est une variable et qu'elle apparait dans l'autre sous-expression (respectivement F' ou E') pour autant que cette dernière ne se réduise pas à une variable
   - alors : pas d'unificateur ; Stop ;
   - sinon : passer à l'étape 6 ;
6. (* Construction d'une substitution de taille un. *)
   (t,X) := un couple (terme, variable) construit à partir de E' et F' ;
7. (* Construction de l'unificateur le plus général. *)
   S := { (t,X) } o S ;
   passer à l'étape 2 ;

Si l'algorithme se termine à l'une des étapes quatre ou cinq, alors les termes "Ti" (i = 1..n) ne sont pas unifiables. Par contre, si l'algorithme s'arrête à l'étape deux, la substitution S est un unificateur de ces termes. En outre, cet unificateur est le plus général. Le lecteur désireux d'obtenir la preuve de cette affirmation la trouvera chez Robinson. [ROBI, 1965]. Nous proposons cependant en annexe 1 un calcul détaillé sur les quelques exemples cités antérieurement.

### 2.2.3. L'unification dans un système Prolog.

Nous n'examinerons pas ici l'implémentation de l'algorithme d'unification d'un interpréteur Prolog car celle-ci dépend fortement de celle des accès aux valeurs des variables. Nous y reviendrons cependant dans un chapitre ultérieur, après avoir examiné ces mécanismes particuliers. Toutefois, nous remarquerons déjà que la plupart des systèmes existant actuellement sur le marché abandonnent le test d'occurrence pour des raisons de rapidité. Ainsi, selon Alain Colmerauer [COLM, 1982], la concaténation de deux listes demande un temps proportionnel au carré de la longueur de la première lorsque le test d'occurrence est implémenté, alors que dans le cas contraire, le temps d'exécution devient linéaire. Cette constatation reste valable avec des algorithmes d'unification dont l'efficacité est particulièrement étudiée.

Hélas, suivant cette option, le système doit s'attendre à devoir traiter dans certains cas particuliers des structures cycliques correspondant à des termes infinis. Si rien n'est prévu, l'interpréteur va au devant d'un double problème :

- l'unification de deux termes infinis peut lancer l'algorithme dans une boucle sans fin;
- la sortie d'un terme infini est impossible.

Actuellement, ces éventualités sont acceptées sur base de la constatation que les cas créant des structures cycliques sont relativement rares, d'autant plus qu'un programmeur avisé se tire aisément de ce genre de situation. D'autres auteurs préfèrent par contre revoir leurs algorithmes pour traiter ces problèmes : [PILG, 1984] , [COLM, 1982] , [HARI et alt., 1984].

# 3. La résolution.

## 3.1. Le principe général.

La résolution est une règle d'inférence conçue pour travailler sur les formules de la logique clausale et permettant de déduire une conséquence de deux clauses possédant une certaine relation. Le principe en est simple et consiste à construire une nouvelle clause par la fusion de deux autres comprenant un même littéral, l'une dans le conséquent, l'autre dans les prémisses. Le littéral dédoublé est également effacé du résultat.

Exemple.

Soient :

Ca1 v Ca2 v ... v Cam :-
Aa1 ^ Aa2 ^ ... ^ Aan avec n,m >= 0. (a).
Cb1 v Cb2 v ... v Cbp :-
Ab1 ^ Ab2 ^ ... ^ Abq avec p,q >= 0. (b).

Supposons qu'un des "Cai" soit identique à un des "Abj" (i = { 1..n }, j = { 1..q }). Alors une conséquence logique des deux clauses (a) et (b) peut s'écrire :

Ca1 v ... v Ca[i-1] v Cb1 v ... v Cbp v Ca[i+1] v ... v Cam :-
Ab1 ^ ... ^ Ab[j-1] ^ Aa1 ^ ... ^ Aan ^ Ab[j+1] ^ ... ^ Abq.

avec (1 =< i =< m) , (1 =< j =< q).

En fait, le principe de résolution est beaucoup plus puissant que ce que pourrait laisser croire cette introduction. En effet, il est parfaitement permis de faire correspondre simultanément plusieurs littéraux des deux clauses avant d'en déduire la conséquence.

D'autre part, nous avons simplifié le principe de correspondance en laissant croire que les littéraux se devaient d'être identiques. En fait, cette identité peut s'établir à une substitution près. Le principe de résolution s'applique alors à l'instanciation des clauses. C'est le procédé d'unification qui permet le calcul de la substitution adéquate.

Les clauses de départ porteront le nom d'hypothèses; toute clause dérivée au moyen de la résolution sera appelée résolvante.

Exemple.

Considérons les clauses :

```
cafardeux(jean) v mauvaisehumeur(jean) :-
                                  jourouvrable(DATE) ^ pluie(DATE).
désagréable(X) :- mauvaisehumeur(X) ^ fatigué(X).
```

signifiant :

- Jean est cafardeux ou de mauvaise humeur les jours ouvrables où il pleut.
- Tout qui est de mauvaise humeur et fatigué se comporte de manière désagréable.

Dans ces deux clauses, les littéraux :

- mauvaisehumeur(jean)
- mauvaisehumeur(X)

sont identiques à la substitution S = { (jean,X) } près. Nous pouvons donc en déduire grâce à la résolution une autre clause :

```
cafardeux(jean) v désagréable(jean) :-
            fatigué(jean) ^ jourouvrable(DATE) ^ pluie(DATE).
```

En effet, de nos deux phrases, nous aurions pu déduire que si Jean est fatigué un jour ouvrable où il pleut, alors celui-ci est cafardeux ou se comporte de manière désagréable.

## 3.2. La réfutation.

Appliquée aveuglément, la résolution ne garantit jamais de déduire une proposition donnée des hypothèses, même si celle-ci en découle logiquement. Il existe cependant fort heureusement un moyen de formuler chaque problème qui permette d'assurer une utilité pratique au procédé : la réfutation.

La méthode consiste à mettre en oeuvre un algorithme de preuve dérivé du "raisonnement par l'absurde" en ce sens que, étant donné un ensemble de clauses "EC" définissant les hypothèses, si nous voulons en déduire un théorème "TH", lui-même représenté par une clause (que nous nommerons le but), le système ne tentera pas de prouver que "EC" implique "TH", mais plutôt que "( ¯TH ^ EC)" conduit à une contradiction ou autrement dit que l'ensemble de clauses { ¯TH , EC } est incohérent.

Est-ce toujours possible? En fait oui. La principale propriété de la réfutation est sa complétude : si un ensemble de clause est incohérent, alors il est toujours possible d'en dériver grâce à la résolution, la clause vide, expression logique d'une contradiction. Par conséquent, si "TH" est un théorème de

"EC", l'ensemble { ~TH , EC } sera incohérent et la réfutation pourra le prouver.

Une seconde propriété de ce procédé est sa cohérence ou correction : il sera tout à fait impossible de déduire une clause vide du système { ~TH , EC } si "TH" n'est pas un théorème de "EC". Cependant, dans une telle éventualité, la réfutation soit échouera soit bouclera infiniment.

## 3.3. Une résolution linéaire sur des clauses de Horn.

Si la réfutation peut toujours déduire la clause vide d'un ensemble de clauses incohérent, cela se fait concrètement en plusieurs étapes impliquant aussi bien les axiomes de départ que les clauses successivement dérivées de ceux-ci par la résolution. Le problème majeur consiste bien entendu à découvrir la séquence suivant laquelle ces clauses doivent être utilisées. Nous nous proposons d'exposer dans cette partie une stratégie particulière que Prolog a adaptée à ses besoins.

En premier lieu, nous limiterons le type de clauses utilisées aux clauses de Horn, tout en regroupant celles-ci en deux catégories : celles possédant une tête et celles n'en possédant pas. Nous remarquerons ainsi que la résolution de deux clauses possédant une tête se conclut toujours en une clause du même type. Par conséquent, la réfutation devra forcément travailler sur un ensemble comprenant au moins une clause sans tête. Trivialement, il s'agit toujours d'un démenti.

La résolution linéaire est un cas particulier de résolution permettant de construire à chaque étape une nouvelle résolvante grâce à :

- la résolvante précédente;
- une clause prise parmi les hypothèses;

et ceci jusqu'à obtenir la clause vide. Au premier cycle, la résolvante est remplacée par le démenti. Puisque ce démenti symbolise le théorème à prouver, nous dirons que la résolution linéaire est d'ensemble support la question posée.

La résolution linéaire avec ensemble support conserve les propriétés de complétude et de cohérence propres au procédé général.

Exemple.

Soit H, l'ensemble des hypothèses :

```
p :- q , r.
q :- s , t.
s :-.
t :-.
r :-.
```

La question est de déterminer si "p" est un théorème de H. Afin de pouvoir appliquer la réfutation, nous ajouterons donc aux hypothèses "¬p", ce qui se représentera au moyen de la clause :

```
:- p.
```

Dès lors, la résolution linéaire s'appliquera de la sorte :

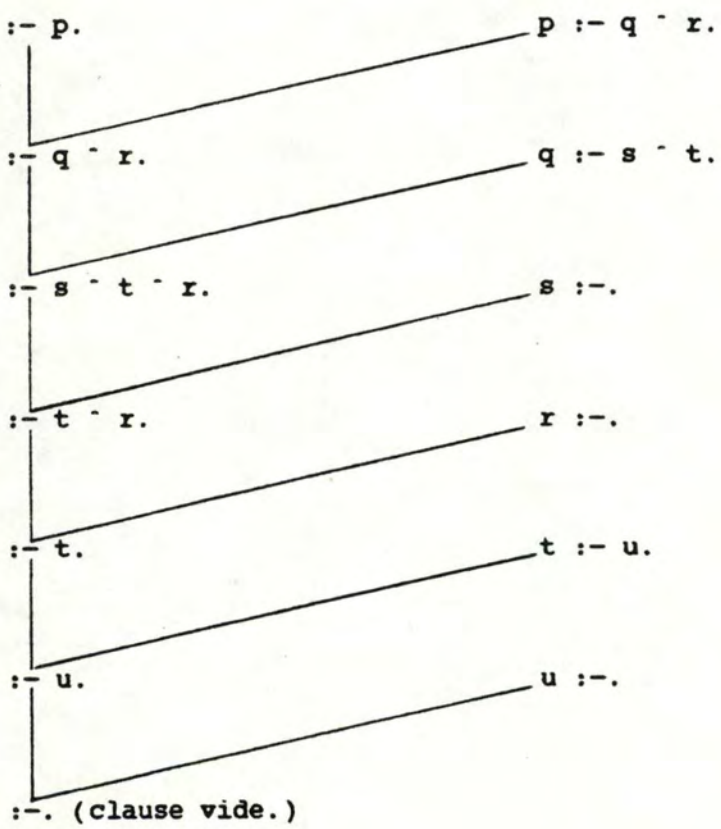



Figure 2.1.

## 3.4. Le cas particulier de Prolog.

Prolog utilise une adaptation particulière du procédé de résolution linéaire sur des clauses de Horn. Il est donc aisé de comprendre la raison pour laquelle toute question au système est posée sous la forme d'un démenti. Celui-ci sera ajouté aux hypothèses constituées par le programme pour ensuite déclencher la résolution.

Le procédé laisse cependant encore une double liberté de choix :

1. la sélection du littéral à effacer au cours de la résolution suivante;
2. la sélection d'une hypothèse pour ce faire.

Cette dualité de choix permet d'affirmer que le système construit un arbre ET/OU, [WINS, 1984] : les noeuds "ET" signifiant que tôt ou tard, chacun des différents littéraux d'une résolvante devra être effacé et les noeuds "OU" représentant l'existence d'hypothèses alternatives pour le faire. Pour ce qui concerne l'avancée, n'importe quelle fonction de sélection peut convenir, pour autant qu'elle assure le choix successif de chaque littéral. Un procédé heuristique d'assurer cette sélection consiste à débuter par le "plus difficile", étant donné qu'en cas d'échec, il n'y aura plus aucune raison de tenter d'effacer les autres. Par contre une seule hypothèse suffisant à effacer un littéral donné, la meilleure manière de procéder est de s'arrêter à la "plus facile" : si elle réussit, la solution la plus simple est découverte.

Prolog n'utilise pas ces méthodes dynamiques, mais au contraire implémente des choix fixes :

1. choix du littéral le plus à gauche dans la résolvante; ce choix est plus généralement désigné sous le nom d'avancée.
2. utilisation des clauses pouvant servir à son effacement dans l'ordre chronologique de leur introduction.

Suivant cette méthode, il se peut très bien qu'à un moment donné l'interpréteur ne puisse plus effacer le littéral sélectionné par l'avancée, aucune hypothèse ne possédant une conclusion s'unifiant avec celui-ci. Dans ce cas, le système reconsidère son choix de clause le plus récent et reprend son raisonnement à partir du point où il était arrivé juste avant celui-ci. Ce mécanisme porte le nom de retour arrière (backtracking).

Afin de pouvoir ainsi revenir sur ses décisions antérieures, le système se ménage des points de reprise ou points de choix associés à un but. Nous dirons qu'il y a possibilité de reprise si, lorsqu'une clause est sélectionnée pour effacer un but, le

paquet restant est non vide. Toutes les clauses permettant d'effacer un but donné font en effet partie d'un même paquet. Il y a échec définif de la question dans l'éventualité où il n'existe plus de point de reprise.

Les systèmes Prolog sont également non-déterministes, c'est à dire qu'il leur est possible, s'ils en sont requis, de fournir toutes les réponses à une question. Le non-déterminisme revient en fait à activer artificiellement le mécanisme de retour arrière après chaque réponse.

Remarquons toutefois que le retour arrière systématique peut être amélioré par l'introduction de procédés permettant d'ignorer certaines tentatives vouées de toute façon à l'échec. Nous en évoquerons quelques-uns dans un chapitre ultérieur.

La stratégie de résolution mise en oeuvre par Prolog correspond dès lors à un parcours en profondeur d'abord et de gauche à droite de l'arbre ET/OU. Bien que le procédé de résolution linéaire soit complet, cette implémentation particulière ne l'est pas : il n'est jamais tout à fait sûr que le système puisse dériver la clause vide d'un ensemble incohérent de clauses.

Ce genre de situation n'est pas toujours facilement détectable. Notre exemple ci-dessous est cependant typique : dans l'une des clauses du programme, il existe une possibilité d'unification entre la tête et un des littéraux du corps. D'autres cas beaucoup plus vicieux peuvent cependant se présenter. Une manière d'éviter ces boucles infinies est de limiter l'interpréteur à une certaine profondeur d'exploration dans l'arbre de résolution.

Exemple.

Soit un programme définissant une généalogie sommaire :

```
gdpère (X , Y) :- père (X , Z) ^ père (Z , Y).
gdpère (X , Y) :- père (X , Z) ^ mère (Z , Y).

mère (X , Y) :- mère (X , Z) ^ frère (Z , Y).
mère (marie , paul) :-.

père (jean , marie) :-.

frère (paul , pierre) :-.
```

et la question :

```
:- gdpère (jean , pierre).
```

pour laquelle la réponse "oui" est attendue. Les résolutions générées par le système sont représentées à la figure 2.2.

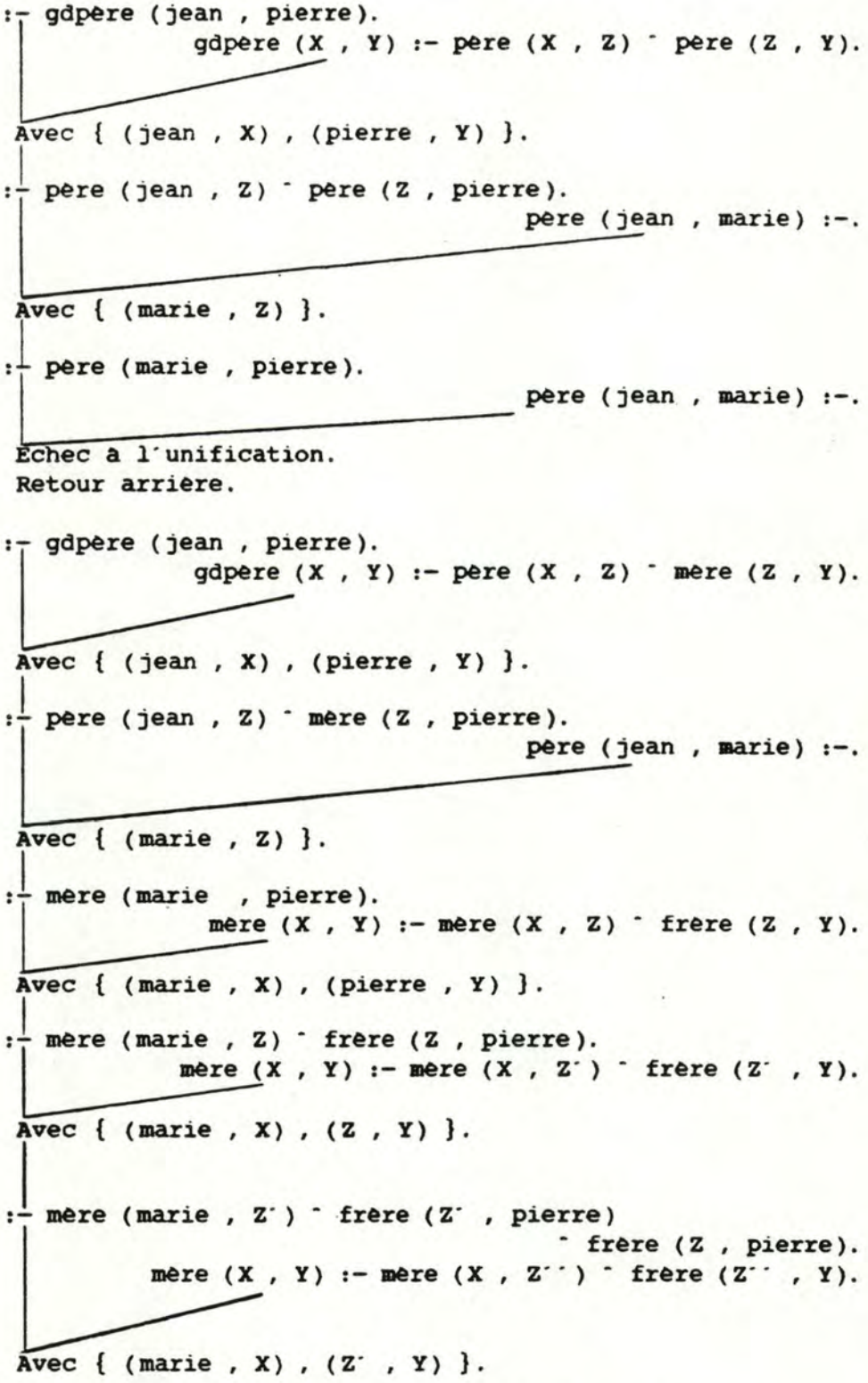

:- gdpère (jean , pierre).
gdpère (X , Y) :- père (X , Z) ^ père (Z , Y).
Avec { (jean , X) , (pierre , Y) }.
:- père (jean , Z) ^ père (Z , pierre).
père (jean , marie) :-.
Avec { (marie , Z) }.
:- père (marie , pierre).
père (jean , marie) :-.
Echec à l'unification.
Retour arrière.
:- gdpère (jean , pierre).
gdpère (X , Y) :- père (X , Z) ^ mère (Z , Y).
Avec { (jean , X) , (pierre , Y) }.
:- père (jean , Z) ^ mère (Z , pierre).
père (jean , marie) :-.
Avec { (marie , Z) }.
:- mère (marie , pierre).
mère (X , Y) :- mère (X , Z) ^ frère (Z , Y).
Avec { (marie , X) , (pierre , Y) }.
:- mère (marie , Z) ^ frère (Z , pierre).
mère (X , Y) :- mère (X , Z') ^ frère (Z' , Y).
Avec { (marie , X) , (Z , Y) }.
:- mère (marie , Z') ^ frère (Z' , pierre)
^ frère (Z , pierre).
mère (X , Y) :- mère (X , Z'') ^ frère (Z'' , Y).
Avec { (marie , X) , (Z' , Y) }.
et le système s'engage dans une boucle infinie ...

Figure 2.2.

# Chapitre 3: Les structures de données de l'interpréteur Prolog.

Dans ce chapitre, nous examinerons l'organisation des structures utilisées par l'interpréteur aux fins de mémoriser les deux catégories de données sur lesquelles il agit : le code source du programme, c'est à dire une représentation interne de celui-ci qui varie très peu durant l'exécution et un jeu de piles beaucoup plus mouvant dont le but est de conserver l'état courant de la démonstration.

## 1. La représentation des clauses en Prolog.

### 1.1. Les termes : une structure d'arbre.

Les objets manipulés par un programme Prolog, en l'occurrence les termes, possèdent une structure tout à fait générale d'arbre.

Un arbre A est constitué d'un noeud particulier R, appelé racine et d'un ensemble ordonné mais éventuellement vide, d'éléments : { A1 , A2 , ... , An } avec n >= 0, qui sont eux-mêmes des arbres.

Les "Ai" (i = 1..n) sont des sous-arbres de A et tout sous-arbre d'un "Ai" est également un sous-arbre de A. Les racines des sous-arbres de A sont les descendants de R. Tout noeud sans descendant est un noeud terminal ou feuille.

Dans la pratique, le symbole fonctionnel d'un terme composé sera représenté par la racine d'un arbre, tandis que les sous-arbres en dépendant symboliseront les arguments. En Prolog, un arbre peut cependant être partiellement inconnu : dans ce cas, une de ses feuilles au moins sera une variable. Les autres noeuds terminaux correspondront aux constantes apparaissant dans un terme composé. Nous pouvons également conclure de cela que tout terme simple est représenté par un arbre réduit à sa racine.

Exemple.

Ainsi, le terme :

- foo (sfct1(Y) , sfct2(7) , a)

peut se représenter au moyen de l'arbre suivant :

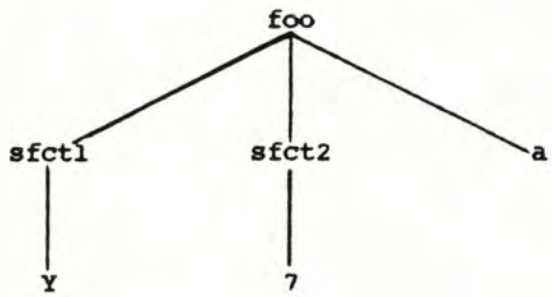



Figure 3.1.

L'efficacité de la structure de représentation des arbres dépend fortement de la manière de mémoriser les identificateurs des symboles fonctionnels, constantes et prédicats. En effet, ces derniers ont une taille en caractères très variable et apparaissent généralement plus d'une fois dans un même programme.

Une façon classique de procéder est de les regrouper dans une zone indépendante, que nous nommerons zone des atomes, dans laquelle tout identificateur n'apparaîtra qu'une seule fois et n'utilisera que le nombre de caractères strictement nécessaire. En outre, une correspondance biunivoque sera établie entre chaque entrée dans cette zone et un atome particulier de manière à garantir qu'une adresse donnée rende toujours un seul et même identificateur et inversement, qu'un identificateur donné ne soit pas enregistré à deux endroits différents de la zone.

Nous pouvons ainsi présenter les grandes lignes d'une structure de données permettant de mémoriser n'importe quel terme. Ainsi, pour un terme composé, celle-ci reprendra au moins les quatre champs suivants :

1. Un drapeau indiquant le type de terme dont il s'agit. La présence de cet élément se justifie par un souci d'efficacité. Il permettra en effet de différencier rapidemment les divers types de termes composés qui apparaîtront au fil des paragraphes suivants.

2. L'adresse du symbole fonctionnel de ce terme dans la zone des atomes.

3. Un entier indiquant l'arité de ce dernier.

4. Un vecteur dont chaque élément aura pour mission de mémoriser un argument. Le champ précédent détermine donc le nombre d'entrées de ce vecteur.

La représentation des termes simples part de la même idée, mais la structure de données peut être nettement simplifiée :

1. Le drapeau indiquant le type de terme est conservé; la distinction suivante sera établie :
   - constante de type atome,
   - constante de type entier,
   - variable.

2. Un second champ dont l'interprétation dépend de la valeur du précédent :
   - pour les constantes de type atome, un pointeur sur l'identificateur dans la zone prévue à cet effet.
   - pour les constantes de type entier, la valeur de cet entier.
   - pour les variables, généralement un entier désignant son numéro d'ordre d'apparition dans le terme dont elle fait partie. La raison d'être de cette numérotation apparaîtra également ultérieurement.

L'implémentation pratique de ces structures peut donner lieu à des variantes justifiées par des soucis d'efficacité.

Exemple.

Le terme :

- foo (sfct1(Y) , sfct2(7) , a)

peut se représenter, par exemple, au moyen de la structure de la figure 3.2.

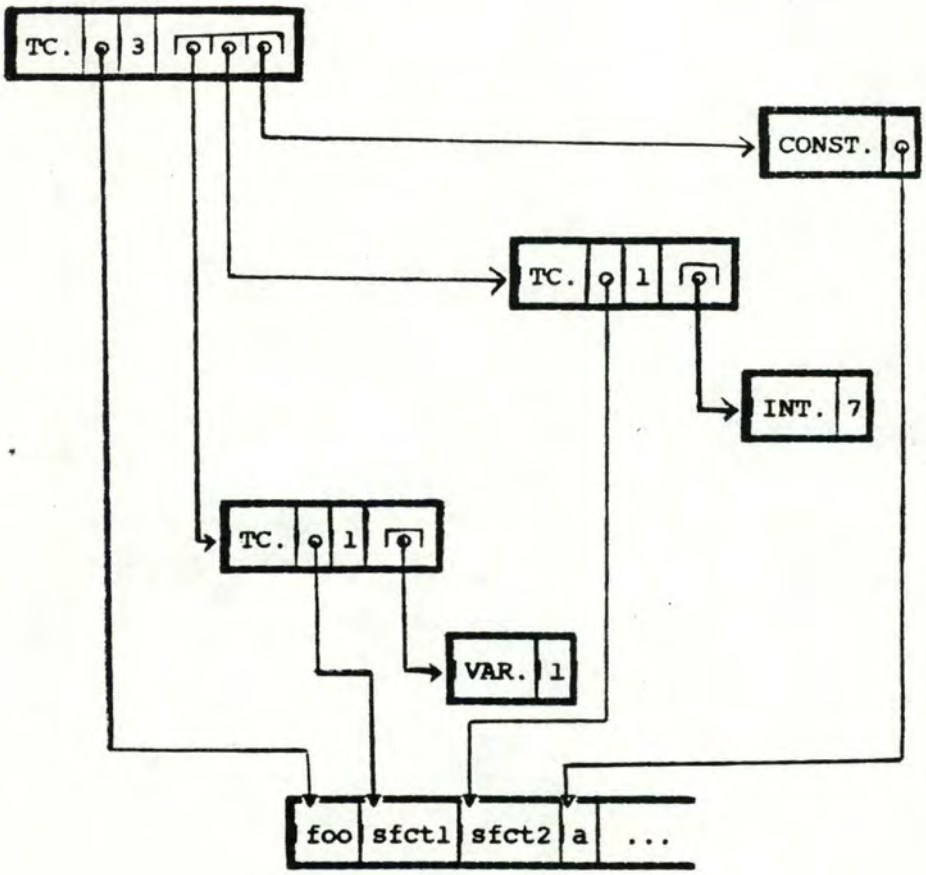



Figure 3.2.

## 1.2. Les clauses : des termes particuliers.

La représentation des clauses en mémoire ne posera en soi aucun problème de part le fait que ces dernières possèdent l'intéressante propriété syntaxique d'être des termes composés. Nous distinguerons les trois cas suivants.

1. Une assertion est formée d'un seul littéral. Or, précédemment, nous avons fait remarquer qu'un littéral pouvait syntaxiquement être considéré comme un terme composé dont le prédicat constituait le symbole fonctionnel. Il suffit donc d'exploiter cette particularité pour les mémoriser.

2. D'autre part, les règles pourront elles-mêmes être considérées comme des termes composés de symbole fonctionnel ":-". L'arité de ce dernier sera deux, son premier argument la tête de la clause et son second argument le corps de celle-ci.

3. Enfin, le corps d'une clause est généralement composé d'une conjonction de littéraux. Il sera dès lors naturel d'envisager un symbole fonctionnel "^", d'arité variant suivant la longueur de la conjonction.

Exemple.

La règle :

```
reverse (cons (X , LA) , LB) :-
 reverse (LA , LC) ^ concatenate (LC , cons (X , nil) , LB).
```

sera représentée par l'arbre :

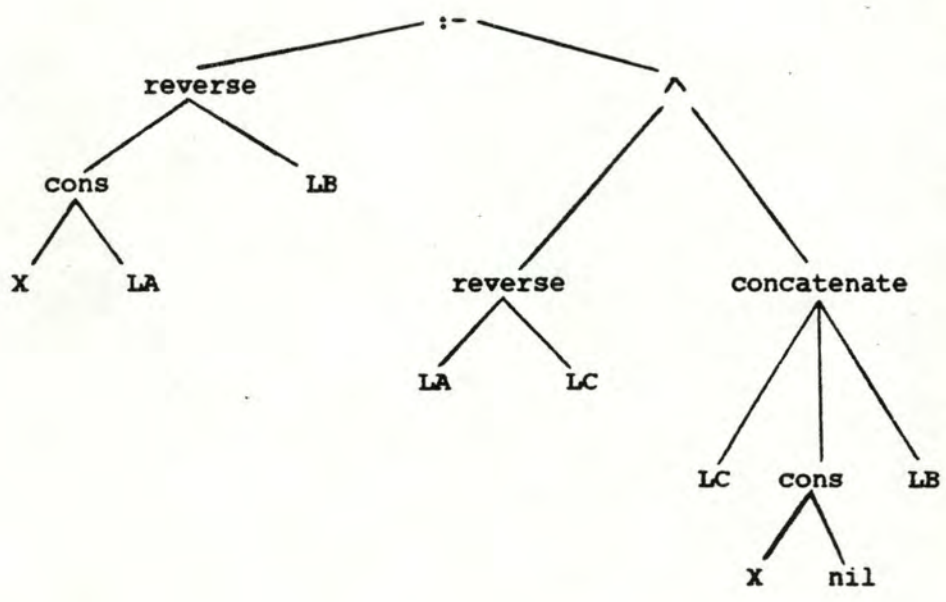



Figure 3.3.

De même, la clause :

```
reverse (nil , nil) :- .
```

sera symbolisée par l'arbre :

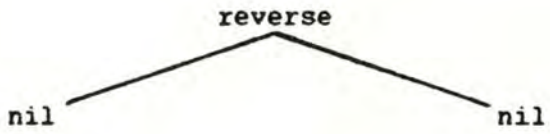



Figure 3.4.

L'avantage d'une telle représentation est que chaque structure, aussi complexe soit elle, est toujours connue uniquement par le biais d'une adresse en mémoire : celle de l'arbre la symbolisant.

## 2. La représentation des instanciations.

### 2.1. L'exemplarisation.

La résolution telle que mise en oeuvre dans un système Prolog fait intervenir le mécanisme d'unification pour instancier les variables des clauses sélectionnées dans le programme.

D'autre part, nous savons que la portée d'une variable Prolog se limite à la clause dans laquelle elle apparaît. Il est par conséquent indispensable de se doter d'un moyen permettant de distinguer les variables de même identificateur utilisées dans des clauses différentes ou résultant de la sélection répétée d'une même clause du programme.

La solution la plus directe consisterait évidemment à recopier chaque clause en y remplacant tout identificateur de variable par un autre, unique. Cependant, ce procédé est à rejeter de par son coût élevé aussi bien en temps de calcul qu'en espace de stockage.

La préférence sera donc donnée à un système utilisant une codification réentrante de telle façon que toutes les utilisations d'une même clause partagent le code la décrivant. Cette méthode peut être mise en oeuvre en utilisant une technique développée par Boyer et Moore [BOYE et alt., 1972] et connue sous le nom de partage de structure, qui consiste à représenter par un couple (squelette, environnement) l'exemplarisation d'une clause. Le squelette est la mémorisation unique de la clause originale tandis que l'environnement est quant à lui un vecteur généré par l'unification et possédant autant d'entrées qu'il n'existe de variables dans le squelette associé. Chaque élément de ce vecteur

permettra de stocker la valeur liée à la variable correspondante.

Avec une telle structure, obtenir l'exemplarisation d'une clause revient à calculer l'instanciation du squelette dans l'environnement.

## 2.2. Chronologie des environnements.

Les environnements possèdent un ordre de création au sein de la démonstration, ce qui va nous permettre de leur associer une chronologie. En effet, l'exemplarisation d'une clause, et par conséquent l'environnement qui lui est associé, est créée lors de l'unification de la tête de celle-ci avec un sous-but à résoudre, lui-même considéré dans un autre environnement. Nous dirons que l'environnement de ce sous-but est antérieur à celui de la clause utilisée par la résolution.

Cette notion de chronologie nous permettra de dater les variables les unes par rapport aux autres, ce qui sera d'un intérêt primordial dans le développement de procédés sophistiqués de récupération de mémoire.

## 2.3. Un exemple explicatif.

Reprenons à ce titre le programme de concaténation de deux listes cité au premier chapitre :

```
concatenate (nil , LA , LA) :- .
concatenate (cons (X , LA) , LB , cons (X , LC)) :-
                              concatenate (LA , LB , LC).
```

Tentons maintenant de prouver le but :

```
:- concatenate (cons (a , cons (b , nil)) , nil , RES).
```

Nous pouvons schématiser la résolution de la manière qui nous est maintenant devenue familière :

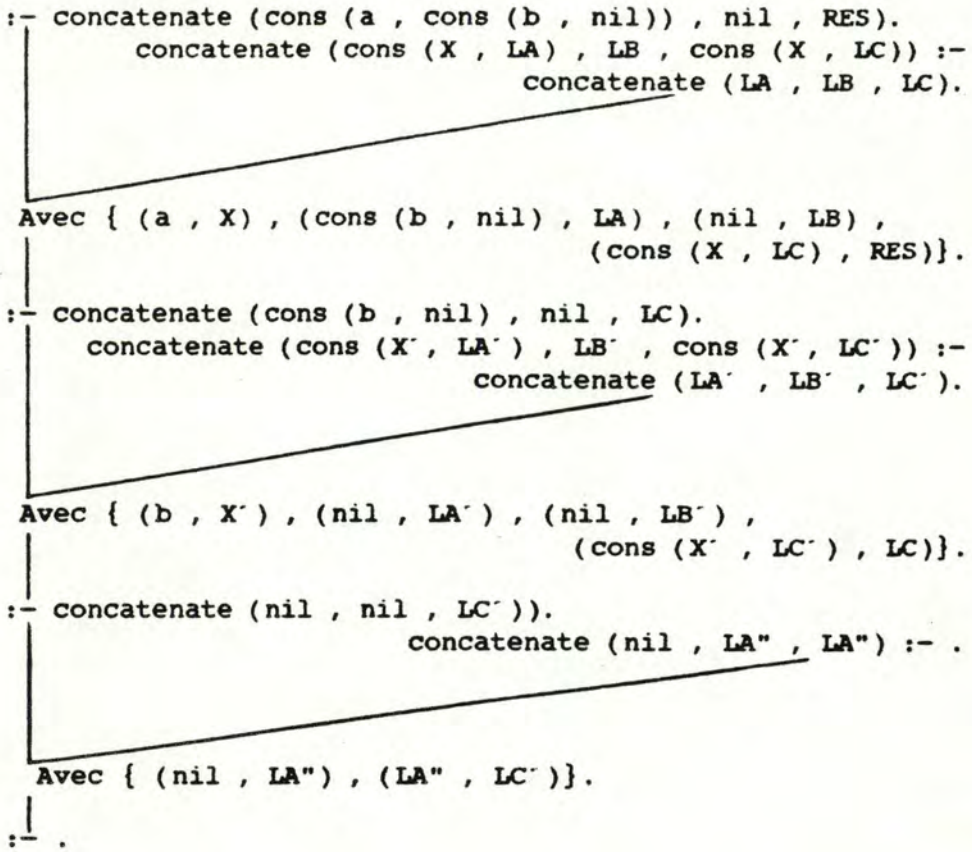



Figure 3.5.

Sur cet exemple, nous pouvons constater que la seconde clause du programme a été utilisée deux fois mais que l'unification a associé lors de chacune de ces utilisations des valeurs différentes aux variables qui y apparaissent. La première clause n'est par contre intervenue qu'une seule fois. Physiquement, trois exemplarisations - outre celle de la question - ont été produites et stockées par le système. Celles-ci sont reproduites à la figure 3.6. où nous lirons "'clause", l'adresse de la représentation physique de la clause.

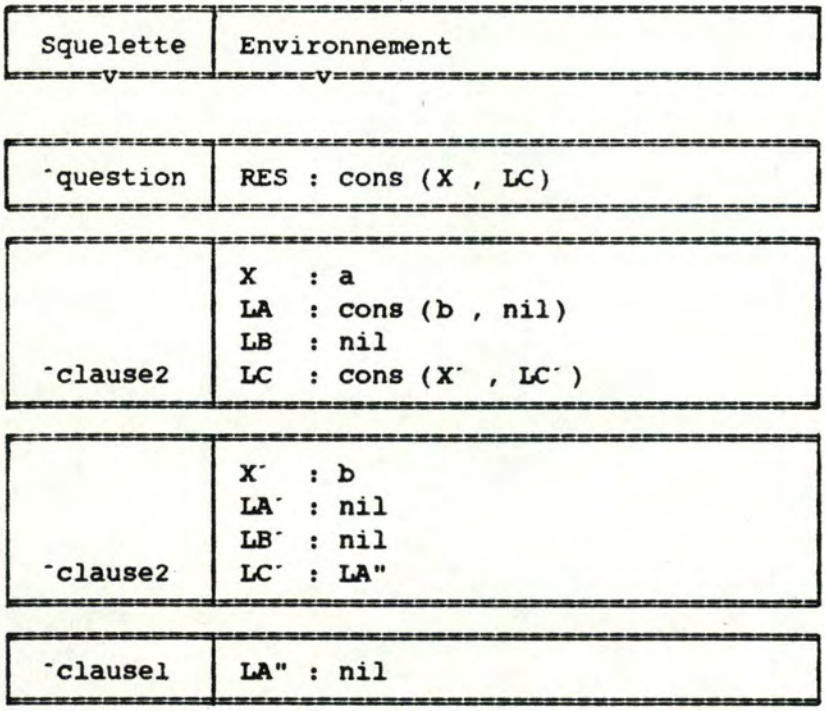



Figure 3.6.

Le lecteur perspicace aura d'ores et déjà compris la raison pour laquelle les variables se voient attribuer un numéro d'ordre dans le code source de la clause : cet entier formera la clé d'entrée dans le vecteur d'environnement, permettant de retrouver très rapidemment la valeur d'une variable.

# 3. Les structures de travail de l'interpréteur.

Dans cette partie, nous nous proposons d'analyser les structures nécessaires à un interpréteur Prolog pour mémoriser l'état courant de la démonstration.

## 3.1. L'algorithme de base.

### 3.1.1. L'arbre de recherche.

Les dérivations successives de résolvantes pratiquées par le procédé de résolution d'un système Prolog peuvent se représenter au moyen d'un arbre "ou", l'arbre de recherche, dont chaque noeud symbolise la conséquence déductible de la résolvante représentée par le noeud père et d'une clause du programme. La question constituera la racine de l'arbre. L'ensemble de tous les chemins ayant la racine pour origine correspond exactement à l'ensemble de toutes les déductions possibles.

Les noeuds terminaux de l'arbre correspondent soit à des clauses vides, soit à des démentis insolubles. La composition de toutes les substitutions effectuées le long d'un chemin menant de la racine à un noeud terminal correspondant à une clause vide, appliquée à la question initiale apporte une réponse à celle-ci.

Exemple.

A titre d'exemple, considérons le programme suivant :

```
p :- q , r.
p :- s , t.
q :- c.
q :- a , b.
a :- .
c :- .
r :- .
```

L'arbre de recherche, à l'exclusion de la représentation des substitutions, produit par la question ":- p." est le suivant :

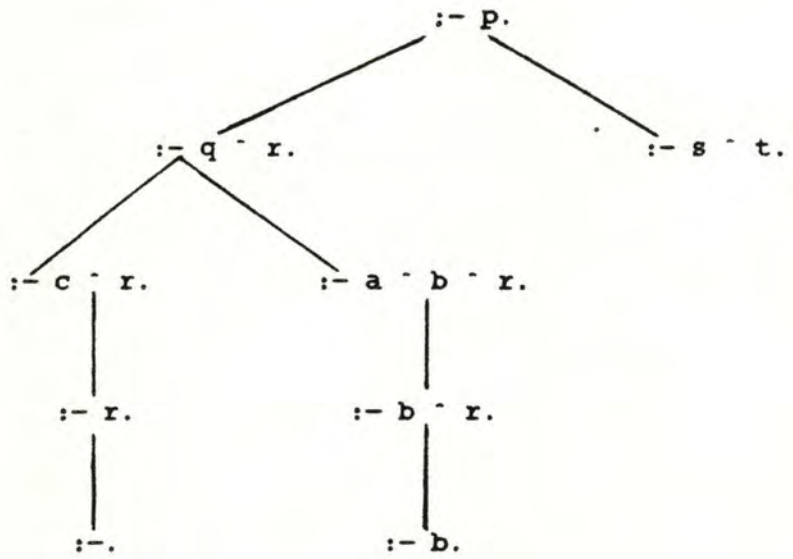



Figure 3.7.

### 3.1.2. Une recherche en profondeur d'abord et de gauche à droite.

Van Emden [EMDE, 1982] obtient un premier interpréteur pour un programme Prolog donné en développant un algorithme qui parcourt l'arbre de recherche en profondeur d'abord et de gauche à droite.

Cependant, pour éviter le calcul exhaustif de cet arbre, celui-ci se donne au préalable une fonction capable de générer successivement toutes les résolvantes déductibles d'un démenti donné, y localisant de la sorte l'implémentation des procédés d'avancée et de choix de clause. Pour la suite de notre exposé, nous nous contenterons d'en reprendre les spécifications :

genfils(d,r) : après avoir été initialisé pour le démenti "d", s'il est possible d'en dériver n résolvantes (n >= 0), les n premiers appels de la procédure renverront la valeur logique "vrai" tout en assignant à la variable "r" une résolvante non encore envisagée tandis que tout appel ultérieur renverra la valeur logique "faux". [EMDE, 1982].

A cette fonction, van Emden associe un mécanisme de pile destiné à mémoriser les résolvantes encore actives,

c'est à dire celles qui sont racines d'un sous-arbre possédant encore au moins une branche incomplètement explorée.

Nous reproduisons cet algorithme ci-après. Dans ce dernier, "rcur" désigne la résolvante courante, c'est à dire la résolvante active la plus récemment atteinte par la recherche.

```
1. (* Initialisations. *)
   Initialiser la pile à vide ;
   rcur := la question initiale ;
   Placer rcur au sommet de la pile ;
2. (* Test de terminaison. *)
   Si rcur est la clause vide
     - alors : Stop avec succès ;
     - sinon : initialiser genfils pour rcur ;
3. (* Exploration d'un nouveau noeud dans la branche. *)
   Si genfils(rcur , x)
     - alors : rcur := x ;
       placer rcur au sommet de la pile ;
       brancher en 2 ;
     - sinon : brancher en 4 ;
4. (* Exploration d'une nouvelle branche. *)
   Si pile vide
     - alors : Stop avec échec ;
     - sinon : retirer le sommet de la pile ;
       le placer dans rcur ;
       brancher en 3 ;
```

Cet algorithme, très simple, peut facilement être généralisé pour produire toutes les solutions.

### 3.1.3. Critique de ce premier interpréteur.

Comme son auteur le reconnaît, cette première solution se trouve être terriblement inefficace car très forte consommatrice d'espace de stockage.

En effet, la pile mémorise une représentation complète de chaque résolvante active, alors qu'une structure de données nettement plus compacte tenant compte de la méthode de résolution utilisée par le système, connue et fixe, pourrait avantageusement être utilisée : chaque résolvante serait alors exprimée en fonction de son ascendante directe et de la résolution intervenue pour sa dérivation.

Cependant, cet algorithme possède également l'interêt de nous faire remarquer que toutes les résolvantes actives sur la pile présentent la particularité de se trouver sur une même branche de l'arbre de recherche ayant pour origine la racine et menant à la résolvante courante. En outre,

l'interpréteur travaille toujours sur cette dernière.

Toutes ces constatations peuvent être exploitées pour définir des structures internes de l'interpréteur nettement plus efficaces. En effet, pour autant qu'il soit possible de définir une structure de données compacte permettant d'exprimer chaque résolvante courante en fonction de son ascendante directe et permettant également de reconstituer cette dernière si besoin en est, seule la résolvante courante doit alors être mémorisée par le système.

C'est une structure présentant de telles propriétés que nous nous proposons maintenant d'examiner.

## 3.2. Une structure plus efficace.

### 3.2.1. L'arbre de preuve.

L'arbre de preuve est une structure de données, en l'occurence un arbre "et", qui stocke sous forme non redondante un chemin dans l'arbre de recherche depuis sa racine jusqu'à la résolvante courante. Ce dernier peut se construire de la sorte :

soit un chemin dans l'arbre de recherche, ch = (R0, R1, ..., Rk) avec k >= 0;

1. si k = 0, alors, le chemin ne comprend que la racine de l'arbre de recherche et l'arbre de preuve s'obtient simplement en attachant à une racine symbolique les différents sous-buts constituant la question initiale.

2. si k > 0, alors notons :

   - SB, le sous-but sélectionné dans la résolvante R[k-1].
   - C, une clause du programme telle que sa tête s'unifie avec SB et dont la résolution avec R[k-1] donne Rk.
   - T, l'arbre de preuve représentant le chemin (R0, R1, ..., R[k-1]) avec k >=1.

   dès lors, l'arbre de preuve représentant le chemin "ch" s'obtient à partir de T, en y attachant comme fils de SB les littéraux formant les prémisses de la clause C et en appliquant au travers du nouvel arbre l'unificateur le plus général de C et de SB.

Exemple.

Voici une séquence d'arbre de preuve correspondant au parcours progressif de la branche centrale de l'arbre de recherche de la figure 3.7.

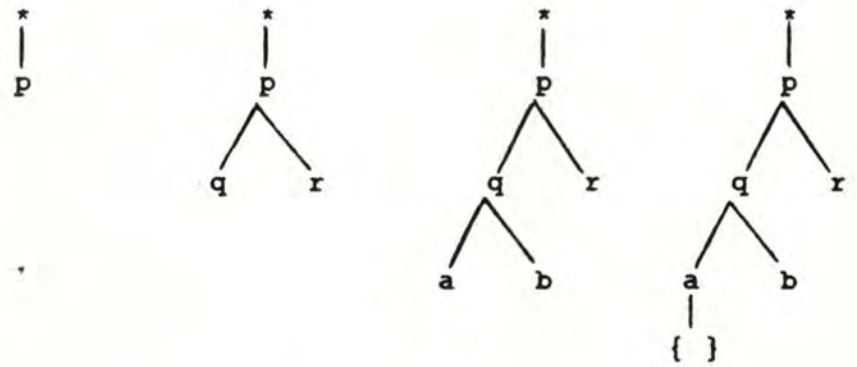



Figure 3.8.

La résolvante courante peut se reconstituer rapidemment grâce aux noeuds terminaux non vides de l'arbre de preuve. En outre si l'ordre de sélection des sous-buts dans une résolvante est connu, il est également très aisé de reconstituer chacun des ancêtres de celle-ci.

## 3.2.2. Implémentation de l'arbre de preuve.

La technique de l'arbre de preuve va permettre la mise en pratique des critiques dégagées au sujet du premier algorithme. En effet, chaque résolution dans l'exécution d'un programme Prolog peut se symboliser par une extension à cet arbre.

Il s'agit cependant de mettre en oeuvre une représentation de ce dernier qui soit suffisamment efficace pour permettre un calcul rapide de tout autre arbre en dépendant. Une solution pratique, proposée par Maurice Bruynooghe [BRUY, 1982], consiste à utiliser une pile dans laquelle demeure toutefois une possibilité d'accès de manière aléatoire. Chaque niveau, désigné ultérieurement sous le nom de bloc d'activation, mémorisera un ensemble d'informations concernant une partie de l'arbre de preuve. Le chemin qu'il représente pourra dès lors être étendu ou contracté très aisément par des ajouts ou des retraits sur cette pile.

Nota bene : Il est absolument indispensable de ne pas confondre cette nouvelle pile avec la précédente dont chaque niveau enregistrait une résolvante complète.

Pour déterminer l'information à placer dans un bloc d'activation, il suffit de considérer la méthode de construction de l'arbre de preuve. Nous pouvons immédiatement remarquer qu'étendre un chemin dans l'arbre de recherche correspond à l'ajout de zéro, un ou plusieurs noeuds dans l'arbre de preuve. En outre, à chacune de ces extensions, correspond une résolution entre la résolvante courante et une instanciation d'une clause du programme.

Dès lors, une solution naturelle consiste à stocker dans chaque bloc d'activation une résolution complète. Pour ce, il suffit simplement d'y mémoriser les instanciations des deux intervenants dans l'unification, puisque les choix de résolution sont connus et fixes.

La représentation de chaque instanciation se fait comme nous l'avons dit au moyen d'un couple (squelette, environnement). Pour mémoriser la clause utilisée, il sera donc créé dans le bloc d'activation un pointeur sur son code source, ce qui permet d'attacher en une seule fois tous les nouveaux noeuds à l'arbre de preuve, et un vecteur d'environnement partagé par tous les noeuds que représentent ces littéraux. L'environnement du sous-but sélectionné dans la résolvante se trouve quant à lui déjà créé dans un bloc d'activation antérieur; nous représenterons donc la seconde instanciation par un couple de pointeurs: l'un sur le code source du sous-but, l'autre sur le bloc d'activation antérieur contenant l'environnement de ce dernier.

En résumé, un bloc d'activation reprendra les quatres champs suivants:

- APPEL : un pointeur sur le code source du sous-but sélectionné dans la résolvante antérieure.
- APPENV : un pointeur sur le bloc d'activation où se trouve l'environnement d'instanciation du précédent sous-but.
- PROC : un pointeur sur le code source de la clause utilisée pour effacer le précédent sous-but.
- PROCENV : un vecteur d'environnement pour cette clause.

Le seul problème à résoudre reste l'environnement de la question, qui peut très bien contenir des variables mais n'est pas exemplarisée lorsque la démonstration démarre. Pour cela, il suffit de considérer une clause bidon :

```
goal :- "question initiale".
```

résolvant la question à un but mais sans variable :

```
:- goal.
```

ce qui justifie la création d'un premier bloc d'activation où le vecteur d'environnement reprend les variables de la question initiale.

Exemple.

Prouvons ":- p(RES)" dans le contexte du programme :

```
p(a) :- q(X , Y).
q(V , U) :- r(U , V).
r(Z ,b) :- .
```

Les résolutions successives qui en découlent sont schématisées à la figure 3.9.

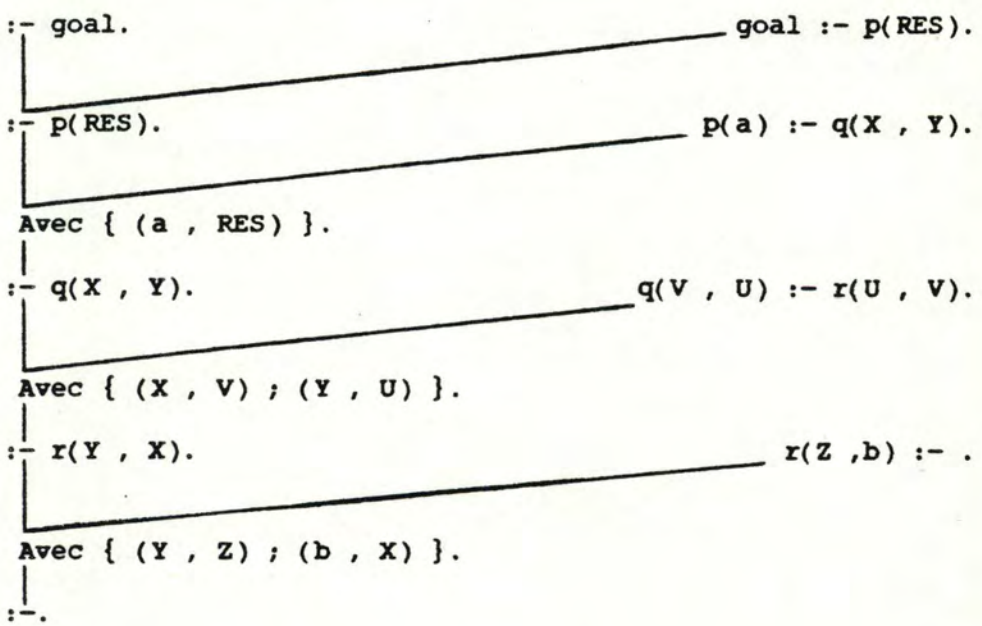



Figure 3.9.

L'évolution de la pile de travail y correspondant est reprise à la figure 3.10.

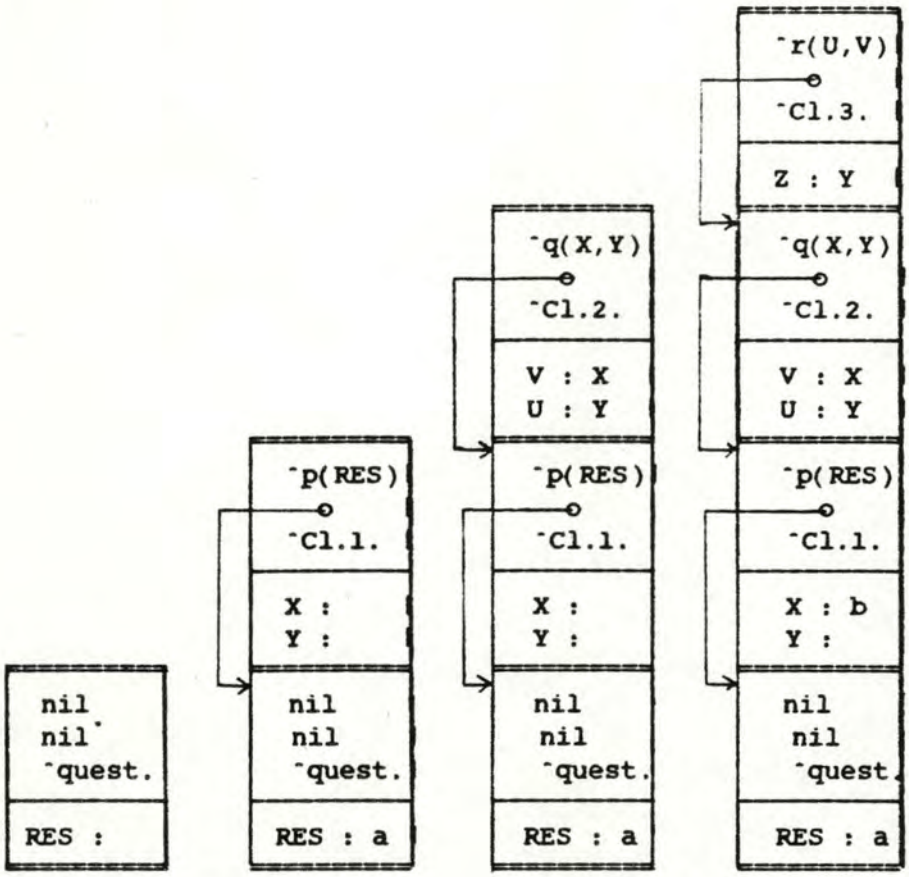



Figure 3.10.

### 3.2.3. Implémentation du retour arrière.

Nous avons volontairement ignoré jusqu'ici l'implémentation du retour arrière dans un but de clarté. Nous nous proposons de maintenant combler cette lacune en présentant les structures nécessaires à la restauration d'une résolvante antérieure.

Comme nous l'avons dit, Prolog explore l'arbre de recherche en profondeur d'abord, travaillant toujours sur la résolvante la plus récemment construite. Par conséquent, toutes les résolvantes pour lesquelles il existe un point de reprise se trouvent sur la branche de l'arbre de recherche menant de la racine à la résolvante courante : ceux-ci peuvent dès lors être mémorisés dans l'arbre de preuve.

Les noeuds progressivement ajoutés à l'arbre de preuve depuis la création du dernier point de reprise forment ce

que nous appelerons le segment courant. En cas de retour arrière, ce segment doit être détruit et les instanciations appliquées durant sa croissance défaites.

Pour ce faire nous introduirons d'abord deux champs supplémentaires dans les blocs d'activation correspondant aux points de reprise:

- BACKTRACK : un chaînage entre les différents blocs de reprise, permettant de retrouver rapidemment le niveau de la pile où revenir en cas d'échec.
- NEXTPROC : un pointeur sur la série de clauses alternatives, ce qui permet de relancer sans trop de frais l'avancée.

Cependant ces informations pourraient facilement être recalculées et ne sont donc pas totalement indispensables.

En fait, le problème majeur posé par le retour arrière provient des instanciations de variables à défaire. Une grande partie des substitutions est détruite avec le segment courant. Cependant, les unifications peuvent avoir modifié certains environnements contenus dans des blocs d'activation antérieurs au dernier bloc de reprise et celles-ci sont nettement moins évidentes à défaire comme en témoigne l'exemple suivant.

Exemple.

Soit le programme :

```
p(X) :- q(X , Y) , r(X).
q(a , b) :-.
q(X , c) :-.
r(c) :-.
```

dans le cadre duquel le but :

```
:- p(a).
```

doit être prouvé. La figure 3.11. montre les résolutions successives qui mènent à la solution et leur effet sur les environnements d'exemplarisation.

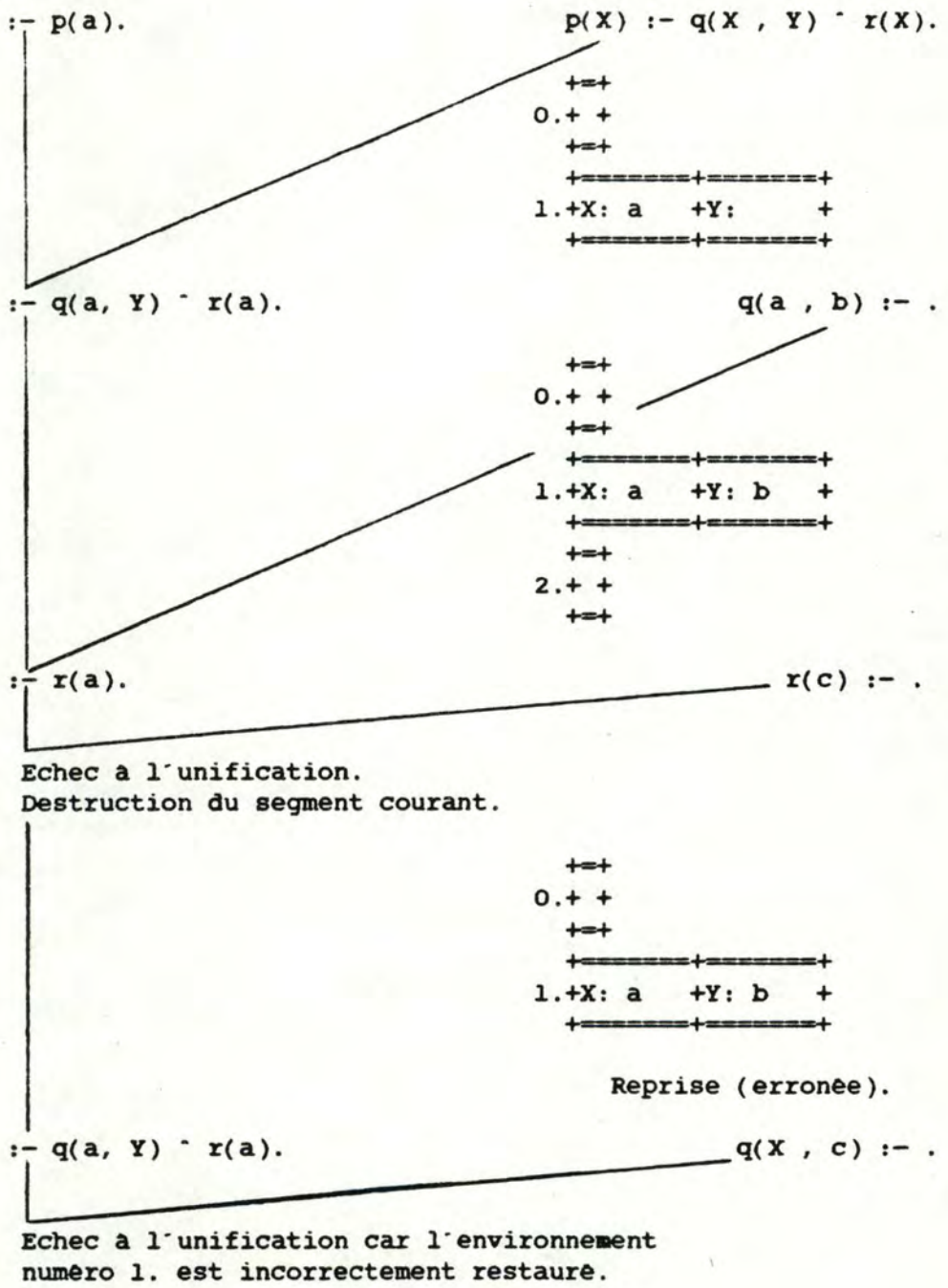



Figure 3.11.

Nous pouvons très bien voir sur cet exemple que l'instanciation de la variable Y est postérieure à la création du bloc de reprise associé à l'effacement du but "q(a Y)". Comme la zone de travail, c'est à dire la pile est globale, cette liaison demeure après la destruction du segment courant car rien ne rappelle au système qu'elle doit

être défaite. Dès lors, la situation se trouve incorrectement restaurée.

Le problème se résoud par l'introduction dans la zone de travail d'une seconde pile appelée pile de restauration (trail) et dont la fonction est de mémoriser la liste des variables instanciées durant la croissance d'un segment auquel elles n'appartiennent pas.

Concrètement, il est assez aisé de distinguer de telles variables. En effet, lorsque l'unification provoque l'instanciation d'une variable, il suffit de tester si la chronologie de celle-ci est antérieure à celle d'un environnement qui est celui du point de reprise à utiliser lors du prochain retour arrière. Si ce test est positif, la variable sera placée sur la pile de restauration. Dans le cas contraire, ce n'est pas nécessaire, puisque l'instanciation sera détruite avec le bloc d'activation la mémorisant lors de la suppression du segment courant.

Par conséquent, lors de chaque retour arrière, la pile de restauration sera consultée afin de libérer les variables qui doivent l'être. Pour cela, une information supplémentaire décrivant l'état de la pile de restauration avant le début de la croissance du segment courant sera introduite dans chaque bloc de reprise. L'état exact de la démonstration avant le choix qui aura provoqué un échec pourra être restauré de la sorte : après avoir détruit le segment courant, le système lira sur la pile de restauration les variables à restaurer et les libérera de leurs instanciations; lorsque cette seconde pile sera revenue à son niveau initial, alors seulement l'avancée sera relancée.

# Chapitre 4: Représentation des valeurs d'une variable.

Après avoir examiné l'organisation générale des structures de données d'un interpréteur Prolog, nous nous proposons d'en détailler un aspect plus particulier, à savoir le mode de représentation de la valeur d'une variable.

## 1. Types d'instanciations.

Nous distinguerons trois types de liaisons d'une variable à une valeur suivant la nature de cette dernière :

- la valeur est un terme simple de type constante;
- la valeur est un terme simple de type variable;
- la valeur est un terme composé.

L'implémentation physique de ces liaisons soulève divers degrés de difficulté. Le premier cas reste cependant trivial : la cellule d'environnement enregistrera simplement la représentation d'une constante telle que nous l'avons présentée dans le précédent chapitre.

Les deux autres cas méritent chacun un examen approfondi.

## 2. Liaisons entre variables.

### 2.1. Liaison de deux variables libres.

Lorsque nous aurons à lier deux variables libres, nous ferons toujours le choix de créer le lien de la plus récente vers la plus ancienne au sens de leurs chronologies respectives. Le respect de cette règle aura pour effet de provoquer les références du sommet de la pile d'environnement vers sa base. Cette propriété n'est absolument pas négligeable car elle permet d'éviter de fastidieuses gestions de pointeurs lors des retraits de blocs d'activation de la pile.

Physiquement, le lien se concrétisera par un pointeur référencant la cellule d'environnement associée à la seconde variable.

Exemple.

Prouvons par exemple :

```
:- p(b).
```

dans le contexte du programme :

```
p(X) :- q(X , Y).
q(V , U) :- r(U , V).
r(Z ,b) :-.
```

Nous obtenons dans ce cas la pile de travail (limitée à la représentation des environnements) représentée à la figure 4.1.

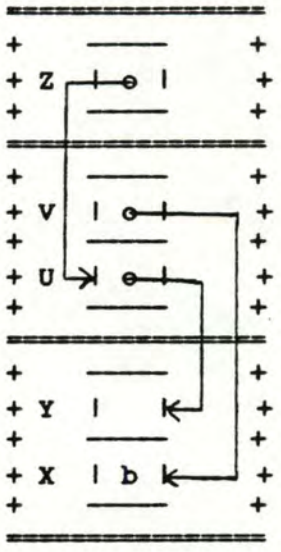



Figure 4.1.

Constatons que chaque bloc d'activation peut se retirer sans jamais provoquer de référence pendante.

L'exemple précédent fait apparaître un important problème dû à cette méthode : les cascades d'indirections. De fait, pour obtenir la valeur associée à la variable Z, il faut accéder successivement à trois environnements. Le problème ne fera que s'amplifier avec la hauteur de la pile. Il semble donc indispensable de mettre au point un procédé permettant de limiter ce genre d'inconvénient.

## 2.2. La déréférenciation.

La déréférenciation consiste à parcourir préalablement à la liaison la chaîne des indirections et d'effectuer ensuite celle-ci directement sur la valeur ultime de la variable. La valeur ultime est celle de la dernière variable de la chaîne si celle-ci est liée ou cette variable elle-même si elle est toujours libre. La cascade d'indirections se verra ainsi drastiquement réduite.

Dans la pratique, le petit algorithme suivant sera appliqué par l'unification lors de chaque liaison entre deux variables.

Pour lier X1 variable libre de E1 à X2 variable de E2,

1. calculer Y, la valeur ultime de X2 ;
2. si Y est une variable libre,
   - alors : si Y est antérieure à X1,
     - alors : créer un lien de Y vers X1 ;
     - sinon : créer un lien de X1 vers Y ;
   - sinon : lier X1 à la valeur Y.

Dans la pratique, le calcul de la valeur ultime d'une variable ne demande plus que rarement le parcours d'une chaîne d'indirections.

Exemples.

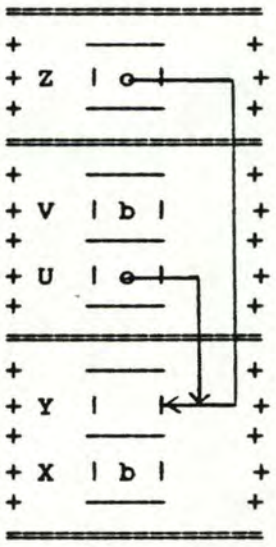



Figure 4.2.

La figure 4.2. représente la pile obtenue pour l'exemple précédent si la déréférenciation es appliquée. Les indirections intermédiaires sur "Z" et sur "V" ont disparu.

Le procédé de déréférenciation ne génera en rien l'implémentation du retour arrière de part le fait que tout lien d'une variable vers une autre est créé du haut de la pile vers le bas. Par conséquent, si une variable doit être libérée de sa valeur au retour arrière, il est certain que toute variable qui aurait pu être instanciée à cette même valeur par le biais de la déréférenciation est également libérée, soit par destruction du segment courant, soit de part sa présence sur le trail.

Remarquons malgré tout que ce procédé ne nous assure pas l'abscence absolue des cascades d'indirections. Ainsi si nous prenons le programme :

```
p(X) :- q(X , Y , Y , a).
q(U , V , U ,U) :-.
```

pour prouver le but :

```
:- p(A).
```

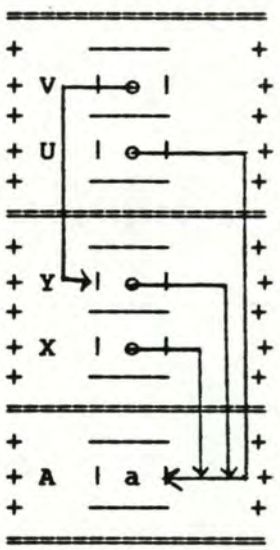



Figure 4.3.

Nous obtenons la pile de la figure 4.3. où demeure une double indirection sur "V", malgré la déréférenciation. Ce cas provient des liaisons intermédiaires entre variables libres auxquelles l'unification n'associe une valeur qu'ultérieurement.

## 3. Liaison d'une variable à un terme composé.

La liaison d'une variable d'environnement à l'instanciation d'un terme composé dépend directement de la manière dont ceux-ci seront représentés en mémoire. Nous nous proposons maintenant d'examiner les deux principales techniques utilisées pour ce faire :

- le partage de structure ;
- la recopie intermédiaire.

### 3.1. Le partage de structure.

Ce n'est pas un hasard si cette première technique porte le même nom que celle utilisée pour la représentation d'une exemplarisation : il s'agit en effet d'une seconde application de la proposition de Boyer et Moore [BOYE et alt., 1972].

Cette solution fut adoptée dès le premier interpréteur, [BATT et alt., 1973] et reste celle des systèmes de David Warren [WARR, 1977] et d'Alain Colmerauer [COLM et alt., 1979].

La cellule d'environnement mémorise ce que Warren nomme une molécule, c'est à dire un couple de pointeurs, l'un vers le code source du terme, l'autre vers un environnement déjà créé où se retrouvent les substitutions à appliquer aux variables du terme précité.

Exemple.

Considérons le programme :

```
p(x) :- q(g(X) , Y).
q(Z , T) :- r(f(Z)).
r(U) :- s(U).
s(f(g(a))) :-.
```

avec lequel nous résolvons le but :

```
:- p(a).
```

L'état final de la pile après les résolutions successives menant à la solution est représenté ci-dessous :

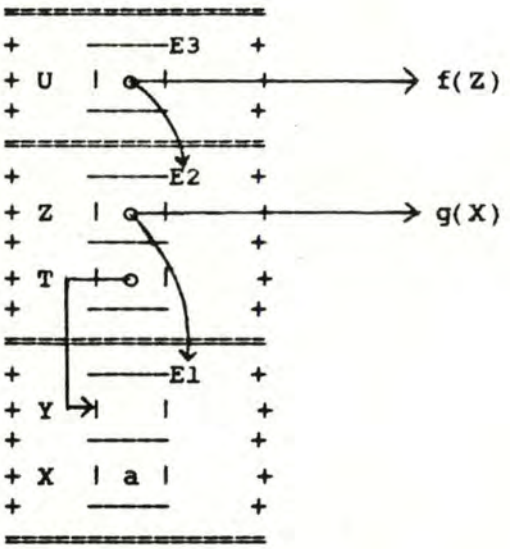



Figure 4.4.

Cette technique permet au travers du mécanisme des liaisons des variables de créer rapidemment de nouvelles instanciations de termes à partir du squelette figurant dans la zone de codage des clauses. La place mémoire nécessaire à la représentation d'un nouveau terme n'est pas fonction de sa longueur.

Par contre l'accès à l'instanciation complète d'un terme ainsi généré est ralenti par la nécessité de parcourir les indirections. Ainsi, dans l'exemple précédent, l'obtention de la valeur de la variable U nécessite l'accès à tous les environnements.

## 3.2. La recopie intermédiaire.

A l'opposé du procédé précédent, la recopie intermédiaire consiste, comme son nom l'indique , à générer une copie littérale de l'instanciation d'un terme lors de sa liaison à une variable. Cette liaison peut se concrétiser par un pointeur vers la représentation de cette valeur, placée dans une zone réservée à cet effet.

Les variables rencontrées lors de la construction d'une copie sont traitées de la sorte, [BRUY, 1982] :

- si la variable est toujours libre dans son environnement, un nouvel emplacement de mémoire est généré et lui est associé. La cellule la représentant dans l'environnement enregistre l'adresse du nouvel emplacement.

- si la variable est dèjà liée à une valeur, un emplacement est également gènèrè et chargè d'une copie de la reprèsentation de cette valeur.

Exemple.

Prenons par exemple le programme suivant :

```
p(x) :- q(X , f(Y)) - r(g(Y)).
q(U V) :-.
r(Z) :- t(Z).
t(g(a)) :-.
```

dans le but de prouver :

```
:- p(a).
```

L'état final de la pile est représenté ci-dessous :

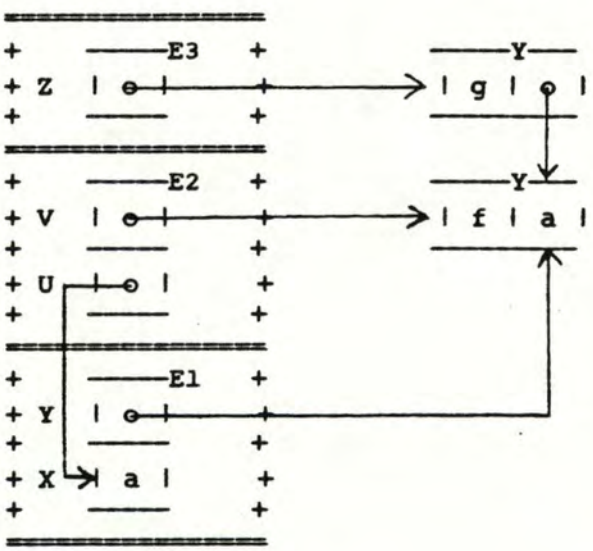



Figure 4.5.

A la différence de ce qui se passe dans le partage de structure, la technique de recopie a pour conséquence que certaines variables se voient associer plusieurs emplacements en mémoire. Tel est le cas dans notre précédent exemple de la variable Y :

- une initiale dans l'environnement d'exemplarisation de la clause où elle apparaît;
- une deuxième lors de la liaison de V à "f(Y)";
- une dernière lors de la liaison de Z à "g(Y)";

Ces locations sont naturellement chaînées entre elles au fur et à mesure de leur création, afin que lorsque Y est enfin liée à une valeur, toutes la désignent. Ce type de phénomène se reproduit à chaque recopie d'un terme où figure au moins une variable non liée.

La technique de la recopie permet cependant une consultation un peu plus rapide du terme; néanmoins, sa création demande plus de temps et d'espace mémoire.

La performance réelle de chacune des deux techniques étant étroitement liée au contexte particulier d'implémentation, nous approfondirons quelque peu leur comparaison après avoir envisagé la mise en oeuvre de certains mécanismes de gestion de mémoire.

# Chapitre 5: Gestion optimisée de la zone de travail.

L'organisation des piles présentée dans le chapitre précédent ne permet une récupération de l'espace de travail que lors du retour arrière. Or, les piles peuvent parfois contenir des informations devenues superflues. C'est le cas notamment au retour des procédures déterministes, c'est à dire n'ayant engendré aucun point de choix. Il en sera de même dans certaines conditions, lors des appels terminaux.

Dans le présent chapitre, nous nous proposons d'abord de passer en revue les méthodes qui ont été mises en oeuvre dans diverses implémentations pour permettre des récupérations anticipées. Celles-ci sont cependant directement influencées par le choix de la technique de recopie ou du partage de structures pour la représentation des instanciations de termes composés. Pour cette raison, nous serons amenés à distinguer explicitement le cas de chacune des méthodes.

Ensuite nous nous attarderons à examiner l'implémentation et l'utilité de "garbage collectors" au sein des interpréteurs Prolog.

## 1. Le retour d'une procédure déterministe.

### 1.1. Que récupérer ?

Lorsqu'un sous-but qui n'a généré aucun point de choix est complètement résolu, sa solution est par définition unique. La branche y correspondant peut par conséquent être élaguée de l'arbre de preuve car elle n'apporte plus aucune information pour la suite de la démonstration. En effet, nous garderons toujours la même résolvante courante et il restera possible de reconstruire toute résolvante correspondant à un point de reprise.

Exemple.

Résolvons ":- p(RES) , t(RES)." dans le contexte du programme :

```
p(a) :- q(X , Y).
p(b) :- q(b , X);
q(V , U) :- r(U , V).
r(Z ,b) :- .
t(X) :- .
```

Les résolutions successives qui en découlent sont schématisées à la figure 5.1.a.

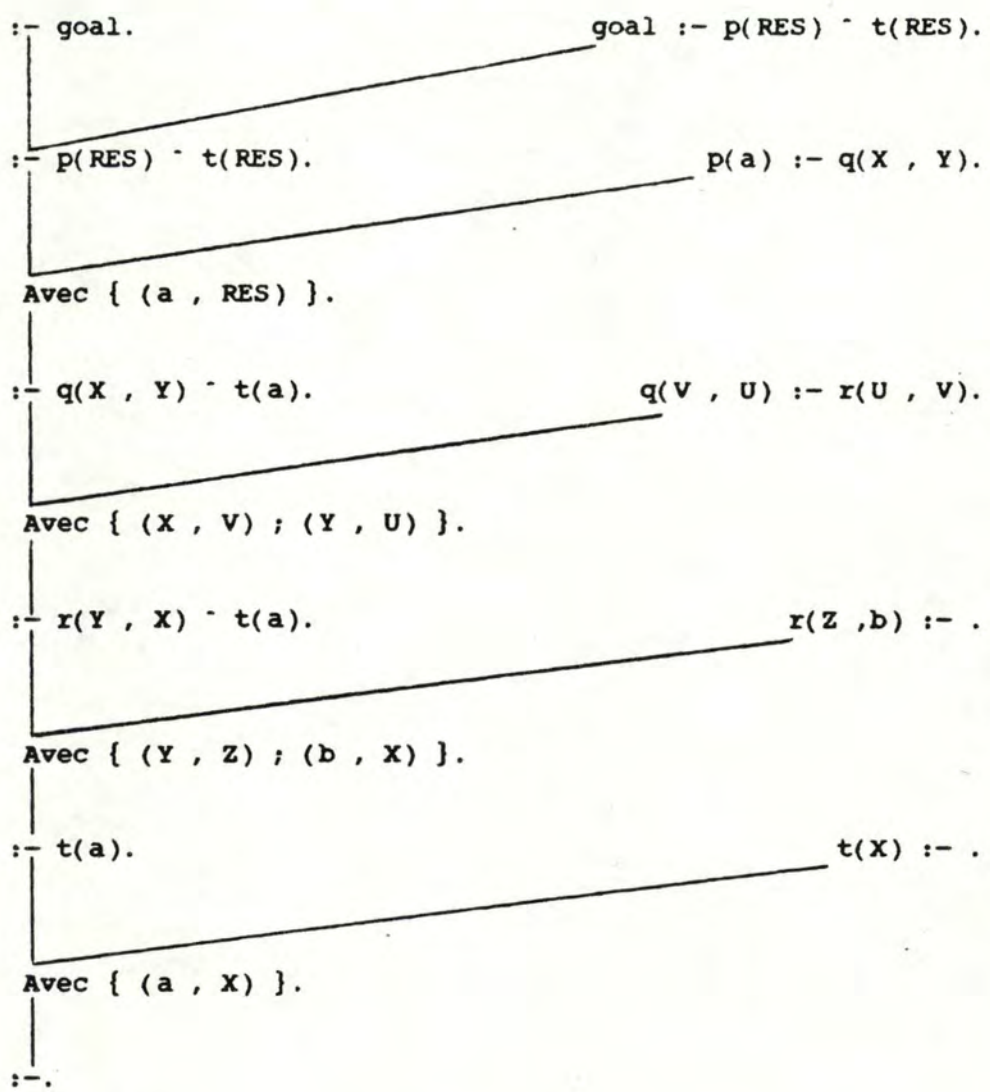



Figure 5.1.a.

L'évolution de la pile de travail y correspondant est reprise à la figure 5.1.b. Lorsque le sous-but "q(X , Y)", déterministe, est complètement résolu, tous les blocs d'activation résultant de sa démonstration peuvent être retirés de la pile de travail sans provoquer de perte d'information pour la suite.

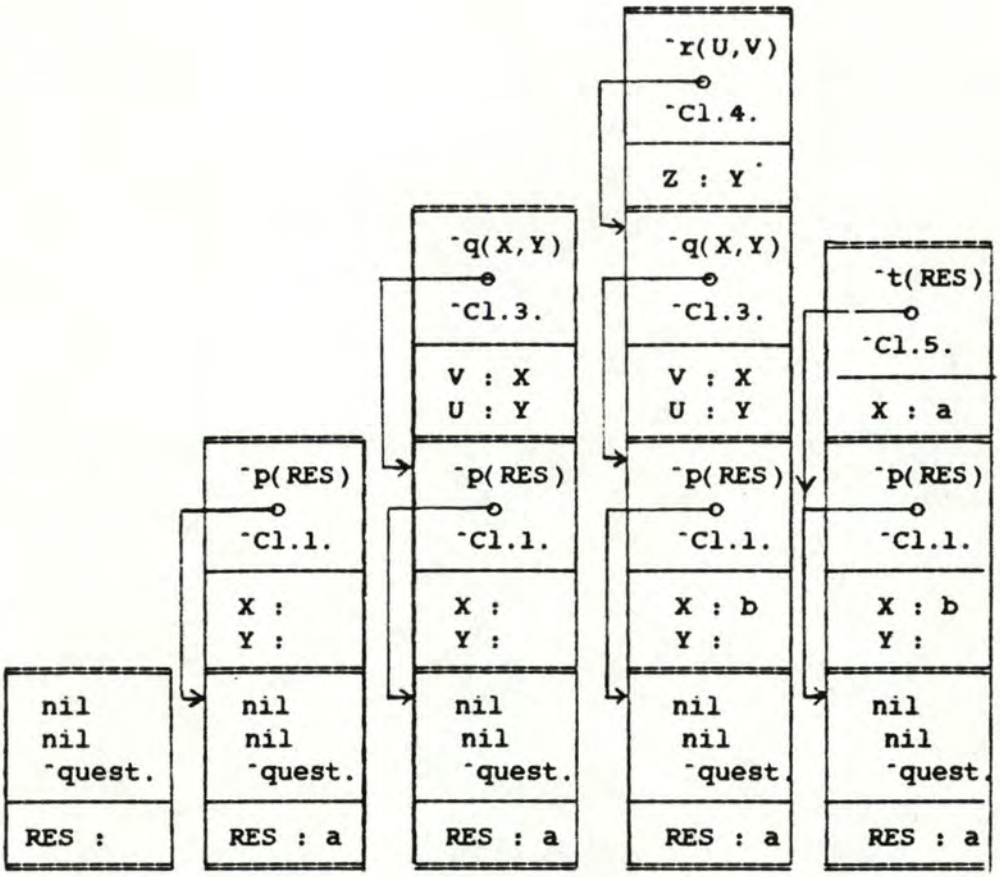



Figure 5.1.b.

## 1.2. Le cas du partage de structure.

### 1.2.1. Un problème particulier.

La méthode d'implémentation des liaisons d'une variable à un terme composé va, dans le cas du partage de structure susciter un problème non trivial lors de la mise en application des méthodes de récupération optimisée d'espace mémoire.

En effet, lorsque qu'une variable "Xi", libre dans un environnement "Ei", devra être liée au terme composé "Tj" instancié dans un environnement "Ej", trois cas devront être distingués, suivant les chronologies respectives de "Ej" et "Ei" :

1. "Ei" est plus récent que "Ej";
2. "Ej" est plus récent que "Ei" mais "Tj" ne contient aucune variable;
3. "Ej" est plus récent que "Ei" et "Tj"" contient au moins une variable.

Si dans le premier cas, le lien créé respecte le sens de l'environnement le plus récent vers l'environnement le plus ancien chronologiquement parlant, il n'en va pas de même pour les deux autres où les liaisons se font à l'inverse. De la sorte, la valeur de la variable sera dépendante d'un environnement postérieur.

Par conséquent, une mise à jour en dehors du retour arrière peut entrainer des pertes d'information dans la section d'environnement de certains blocs d'activation, pertes qui s'extérioriseront par le biais de références pendantes.

Exemple.

Aux fins d'illustrer une telle situation, considérons le programme suivant :

```
p(X) :- q(X ,Y) ^ s(X) ^ r(Y).
q(a , f(Z)) :- t(Z).
t(b) :-.
s(a) :-.
r(f(b)) :-.
```

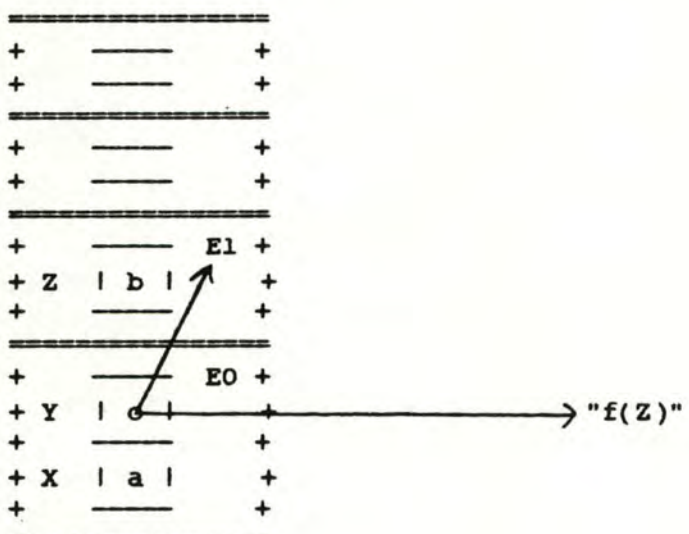



Figure 5.2.

La pile de travail (limitée à la représentation des environnements) issue de la démonstration de ":- p(a)" est symbolisée à la figure 5.2.

Les exécutions de "p", "q", "s" et "r" sont totalement déterministes. Cependant, au retour de la procédure "q", son bloc d'activation n'aurait pu être supprimé en raison de l'allocation d'un pointeur du bloc antérieur vers son environnement, auquel cas une référence pendante aurait été produite.

### 1.2.2. La solution "Colmerauer".

Pour résoudre ce problème, une solution assez directe consiste à scinder les blocs d'activation de façon à en exclure le vecteur d'environnement. La zone de travail s'organise dès lors en trois piles:

- La pile de restauration, dont le contenu reste celui qui a précédemment été défini et n'est mis à jour que lors des retours arrière.
- Une pile d'environnements, également mise à jour dans le seul cas d'un retour arrière.
- La pile de travail, mise à jour lors des retours arrière mais aussi au retour des procédures déterministes.

Cette solution, proposée par Alain Colmerauer [COLM et alt., 1979], manque cependant quelque peu de finesse : elle maintient indistinctement sur la pile des environnements les liaisons nécessaires à la poursuite de la démonstration et celles qui ne le sont plus. Cet inconvénient a amené David Warren à envisager une seconde méthode.

## 1.2.3. La solution "Warren".

### 1.2.3.1. Classification des variables.

David Warren [WARR, 1977] commence par classer les variables figurant dans une clause suivant le temps où peut être utilisée l'information qu'elles portent. Il aboutit ainsi à en distinguer quatre types :

1. Les variables globales, qui peuvent être référencées à tout moment de la démonstration. Elles possèdent toujours au moins une occurence au sein d'un terme composé.

2. Les variables non globales, c'est à dire ne possédant pas d'occurrence au sein d'un terme composé, qui se répartissent quant à elles en trois catégories :

   - Les variables passe-partout, qui n'apparaissent qu'une seule fois dans la clause. Elles satisferont toute unification mais ne seront jamais réutilisées par la suite.

   - Les variables temporaires apparaissent plusieurs fois dans la clause mais en tout cas jamais dans les antécédents. Leur utilité ne s'étend pas au-delà de l'unification de la tête de la clause avec un sous-but à effacer.

   - Les variables locales pouvant elles aussi apparaître plusieurs fois dans la clause mais au moins une fois dans un des antécédents. Ces variables sont utilisées tout au long de la démonstration de la clause dont elles font partie mais jamais après la terminaison de celle-ci.

Exemple.

Dans la clause :

```
p(f(X) , Z , Y , Z , R) :- q(X , R) , r(R).
```

les variables "X", "Y", "Z" et "R" sont respectivement globale, passe-partout, temporaire et locale.

### 1.2.3.2. Gestion de mémoire associée.

De cette classification découle un certain nombre de conséquences.

En premier lieu, il est absolument superflu de mémoriser la valeur des variables passe-partout et donc de leur réserver un emplacement dans l'environnement.

Les variables temporaires ne seront mémorisées que durant le déroulement de l'unification au cours de laquelle elles interviennent. Dès la terminaison de celle-ci, elles seront éjectées puisque leurs valeurs ne sont pas nécessaires à l'interprétation des appels du corps de la clause.

Les variables succeptibles d'intervenir dans les liaisons à des termes composés du troisième type précité peuvent aisément être déterminées : ce sont les variables globales.

David Warren propose d'isoler ces variables sur une pile afin de pouvoir les gérer distinctement des autres. Ainsi chaque environnement se trouve scindé en deux parties, l'une dite globale, l'autre locale, où les variables sont réparties suivant leur nature.

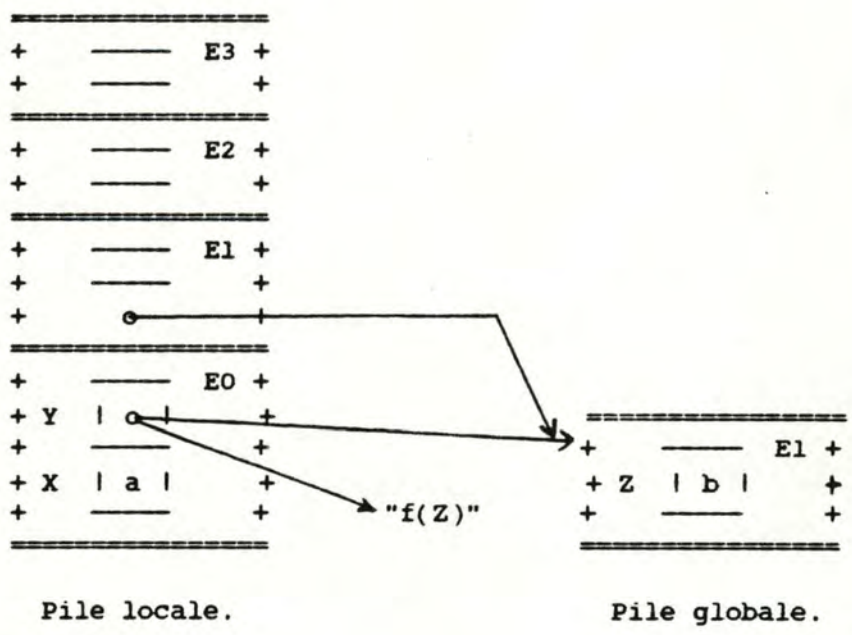
Dans cette nouvelle organisation, la création d'un bloc d'activation est remplacée par la création d'un bloc sur chacune des piles locale et globale. Le bloc local reprendra toute l'information que contenait auparavant le bloc d'activation, à l'exclusion de la partie globale de l'environnement remplacée quant à elle par une référence au bloc qui la mémorise sur l'autre pile.

Exemple.

Ainsi, lorsque ":- p(a)." est prouvé dans le cadre du programme :

```
p(X) :- q(X ,Y) ^ s(X) ^ r(Y).
q(a , f(Z)) :- t(Z).
t(b) :-.
s(a) :-.
r(f(b)) :-.
```

nous obtenons désormais les deux piles reprises à la figure 5.3.




Pile locale. Pile globale.

Figure 5.3.

Avec une telle organisation, la mise à jour de la pile locale devient possible lors de tout retour d'une procédure déterministe, puisque les références s'y font désormais toujours soit du haut vers la bas, soit vers la pile globale, à laquelle rien n'est retranché si ce n'est dans le cas normal d'un retour arrière.

### 1.2.3.3. Un perfectionnement complémentaire.

Toutes les variables globales ne génèreront cependant pas nécessairement une référence du bas vers le haut de la pile de travail. En effet, les termes composés sont utilisés soit pour décomposer un terme déjà existant, soit pour en construire un nouveau.

Ainsi par exemple la définition de la concaténation de deux listes déjà énoncée et que nous reprenons ici :

```
concatenate (nil , LA , LA) :- .
concatenate (cons (X , LA) , LB , cons (X , LC))
                    :- concatenate (LA , LB , LC).
```

peut être utilisée de deux façons distinctes, à savoir le calcul de toutes les sous-listes d'une liste donnée ou le calcul de la concaténation de deux listes.

Dans le premier cas, par exemple :

```
:- concatenate(L1 , L2 , (a.b.c)).
```

le terme

```
cons (X , LA)
```

en construit de nouveaux au fur et à mesure des exemplarisations, ceci par le biais des liaisons incriminées. Au contraire, le terme

```
cons (X , LC)
```

permet la décomposition de la liste initiale "(a.b.c)".

Dans le second cas, par exemple :

```
:- concatenate ((a.b.c) , (d) , RES)
```

les rôles sont inversés,

```
cons (X , LA)
```

décomposant la liste initiale et

```
cons (X , LC)
```

créant la concaténation.

Par conséquent, dans le premier cas il aurait suffi de placer X et LA sur la pile globale, bien que LC soit également globale. Dans le second, seules les

variables X et LC devraient aller sur la pile globale. En fait, la classification statique de Warren amène à considérer comme globales des variables qui se comportent dans certains cas comme des variables locales.

C'est la raison pour laquelle, ce dernier introduit dans son implementation une option de déclaration permettant à l'utilisateur de spécifier au besoin l'utilisation qui sera faite des arguments d'une procédure. L'interpréteur considérera alors comme locales certaines variables qui auraient été placées sur la pile globale.

Evidemment, ce procédé possède l'inconvénient de supprimer la réversibilité caractéristique d'un programme Prolog. C'est sans aucun doute la raison pour laquelle la déclaration de mode n'a aucun caractère coercitif.

## 1.3. Le cas de la technique de la recopie.

Le fait de générer des copies des termes composés dans une zone indépendante de la pile de travail résoud d'emblée le problème qui se présentait dans le cas du partage de structure. En pratique, la zone des recopies est elle-même organisée sous forme de pile. La liaison d'une variable à un terme composé se traduit donc toujours par une référence vers cette autre pile.

Exemple.

Si nous résolvons la question ":- p(a)." dans le cadre du programme :

```
p(X) :- q(X ,Y) ^ s(X) ^ r(Y).
q(a , f(Z)) :- t(Z).
t(b) :-.
s(a) :-.
r(f(b)) :-.
```

nous obtenons au retour de la procédure déterministe "q", l'état des deux piles symbolisé à la figure 5.4.

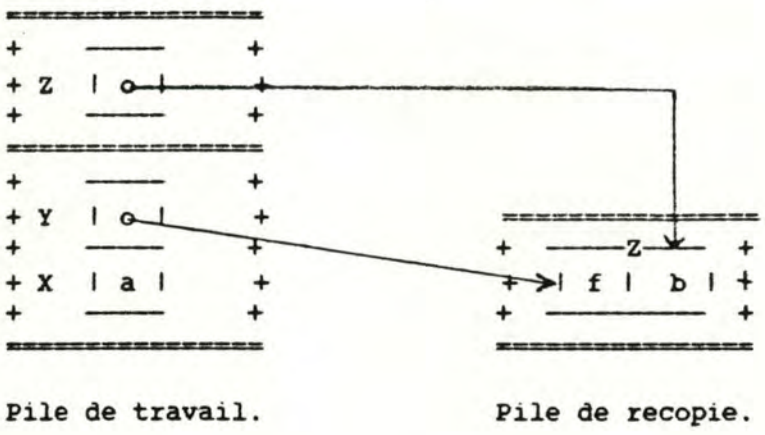



Figure 5.4.

Rien n'empêche la récupération du bloc de la pile de travail généré par cette procédure déterministe. Dans l'exemple précédent, nous obtiendrions après cette opération l'état des piles représenté à la figure 5.5.

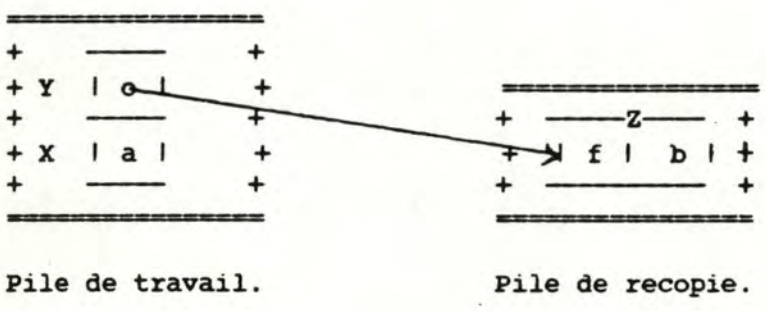



Figure 5.5.

## 1.4. Performances comparées des deux méthodes.

Comme nous l'avons déjà évoqué, la technique du partage de structure devrait se comporter plus efficacement lors de la construction d'instanciations de termes composés alors qu'inversement, pour ce qui concerne l'accès à ceux-ci, l'avantage reviendrait à la recopie intermédiaire. En effet, en partage de structure, chaque variable n'est représentée qu'une seule fois tandis que la recopie intermédiaire associe à chaque occurrence d'une variable dans un terme composé soit une nouvelle location, soit la copie d'un sous-terme instancié. De la sorte le

partage de structure semble consommer moins d'espace sur la pile globale que l'autre méthode n'en nécessite sur la pile de recopie. En contrepartie, l'accès à un terme déjà construit en partage de structure se trouvera ralenti par de plus nombreuses indirections.

Nous pouvons dès lors en conclure que l'utilisation d'espace mémoire par les piles globale et de recopie dépendra essentiellement des programmes qui seront soumis à l'interpréteur. Le partage de structure se montrera en principe plus performant avec un programme construisant sans cesse d'énormes termes composés sans jamais y accéder tandis que la recopie intermédiaire l'emportera dans le cas inverse. Il faut cependant s'attendre à ce que les programmes "réels" se situent entre ces deux extrêmes.

Pour ce qui concerne la pile d'environnement, le partage de structure l'emporte très certainement la plupart du temps. En effet, cette technique n'y mémorise que les variables locales tandis que la recopie intermédiaire y reprend également les variables globales. Cependant, comme la pile d'environnement peut être réduite au retour des procédures déterministes, la recopie intermédiaire sera sans aucun doute avantagée dans le cas de programmes hautement déterministes.

Au cours de la discussion ci-dessus, nous avons toujours présumé qu'il était possible de mémoriser une molécule, c'est à dire un couple d'adresses, dans l'espace réservé pour la valeur d'une variable, à savoir le plus souvent la plus petite unité d'espace adressable sur la machine : le mot. Christopher Mellish, [MELL, 1982], fait judicieusement remarquer que si la taille de ce dernier est faible, il devient impensable d'y caser deux adresses !

En fait, ce problème ne peut être négligé car il se pose très fréquemment. Dès lors, il est bon d'examiner les possibilités de le résoudre.

1. La première solution est brutale : il suffit de réserver une entité plus grande pour la valeur d'une variable, par exemple deux mots de mémoire.

2. Une alternative est de gérer séparément les molécules sur deux mots en les allouant sur la pile globale et en les référencant par le biais de la cellule réservée à la valeur de la variable.

Etant donné que nombre de variables ne nécessitent pas l'emploi d'une molécule, la première solution est assez dispendieuse. En effet, supposons un appel nécessitant la réservation d'espace pour k variables. Dans le premier cas, 2*k mots seront utilisés, car à priori, il est impossible de déterminer les variables dont la valeur sera stockée sous forme de molécule des autres. Par contre le second cas ne réservera que k mots et par la suite, pour chaque molécule générée, 2 mots supplémentaires. Posons le

problème sous forme d'équation :

$2 * k >= k + 2 * x$ où x représente le nombre de molécules.

$k >= 2 * x$

$x =< k/2$

Ce qui signifie que dans la pratique, la seconde solution est plus écomonique tant que au plus une variable sur deux voit sa valeur mémorisée sous forme de molécule. Christopher Mellish précise que c'est le cas de tous les essais qu'il a pratiqués [MELL, 1982]. Après examen de ses programmes d'essai, nous pouvons raisonnablement conclure que la probabilité de l'autre situation est minime.

Suivant cette méthode, l'utilisation de la pile globale va se trouver accrue. Par conséquent, il est très plausible que, à programme identique, celle-ci soit plus volumineuse qu'une pile de recopie.

Nous pouvons conclure cette comparaison par une double remarque :

- Comme nous pouvions nous y attendre, l'avantage d'une méthode sur l'autre n'est pas tranché.
- D'autre part, il est intéressant de constater qu'un facteur significatif de comparaison soit la relation entre la taille d'un mot de mémoire et celle de l'adresse de celui-ci.

## 2. Transformation de l'appel terminal.

La gestion spécifique de l'appel du dernier antécédent d'une clause va également permettre une réduction de la consommation de l'espace de travail. En effet, le bloc d'activation associé à une procédure déterministe sera récupéré non plus cette fois au retour de celle-ci mais déjà lors de l'activation de son dernier littéral.

### 2.1. Origine de l'idée.

Les interpréteurs Lisp et notamment ceux développés à l'université de Vincennes (Vlisp) par Patrick Greussay, [GREU, 1977], gèrent de manière itérative certains types de récursions. Il ne s'agit pas ici d'une transformation syntaxique ou sémantique au sens où le proposent des auteurs comme Burstall et Darlington, [BURS et alt., 1977], visant à obtenir la version itérative d'un programme récursif, mais bien d'une gestion

spéciale des ressources nécessaires à l'évaluation. Les récursions terminales sont traitées comme des itérations et leur exécution s'effectue en espace constant.

## 2.2. Transformation de l'appel terminal en Prolog.

### 2.2.1. Conditions d'application.

L'application du procédé est subordonnée à la vérification préalable de deux conditions :

1. le littéral doit occuper la dernière position dans le corps de la clause.

2. aucun point de choix n'a été généré pour les littéraux précédents dans le corps.

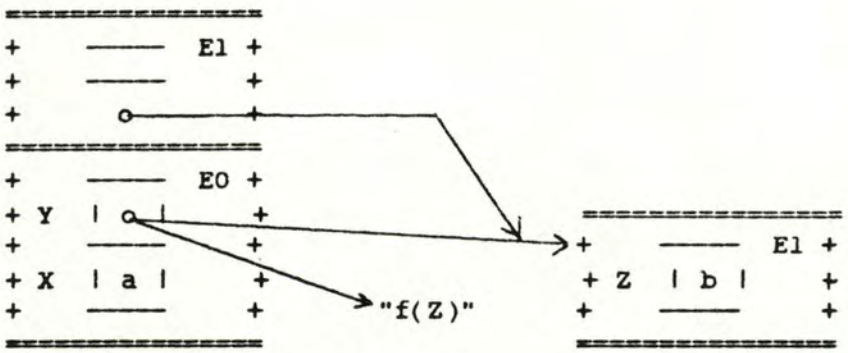
La mise en oeuvre de cette transformation est du reste possible aussi bien dans le cas de la représentation des termes composés par partage de structure que par recopie intermédiaire. Cependant, l'application du procédé dans le cas du partage de structure se heurte au même problème que celui exposé pour la récupération des blocs déterministes. La même solution restant d'application, nous ne distinguerons plus explicitement les deux implémentations.

Exemple.

Reprenons l'exemple de la section précédente :

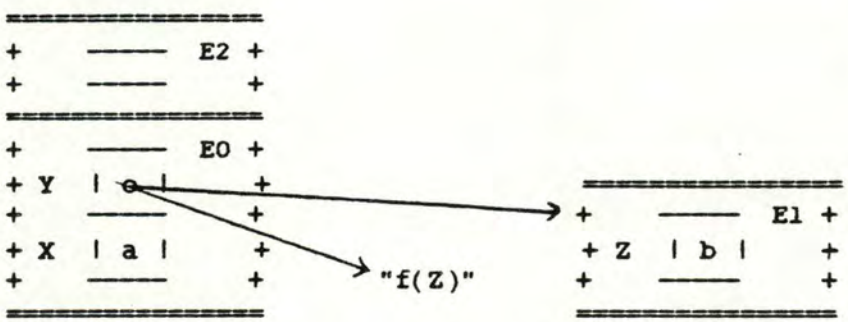
```
:- p(a).
p(X) :- q(X ,Y) ^ s(X) ^ r(Y).
q(a , f(Z)) :- t(Z).
t(b) :-.
s(a) :-.
r(f(b)) :-.
```

La figure 5.6 montre la récupération qui s'effectue dès l'appel de "t(Z)", terminal dans la définition de la clause "q". Le bloc d'activation associé à cette clause est directement remplacé par celui dont la création est consécutive à l'appel de la procedure "t".




Pile locale. Pile globale.

Avant la transformation.




Pile locale. Pile globale.

Après la transformation.

Figure 5.6.

## 2.2.2. Précautions d'implémentation.

L'implémentation de ce procédé nécessite toutefois la mise en oeuvre de certaines précautions.

Tout d'abord, les valeurs des variables apparaissant dans le dernier littéral doivent être sauvegardées afin de pouvoir mener à bien son exécution. Le cas des variables locales non liées lors de cet appel pose problème puisque leur emplacement mémoire a disparu. La solution proposée par David Warren, [WARR, 1980] est de rendre ces variables globales.

D'autre part, le procédé peut également fausser l'effet du "cut", un prédicat particulier de Prolog ayant pour but de rendre déterministe la procédure au sein de laquelle il est exécuté.

Exemple.

Considérons la situation décrite par la figure 5.7. où nous noterons :

- r : la racine de l'arbre de preuve,
- bi : les blocs d'activation correspondant à des points de choix,
- di : les blocs d'activation déterministes et,
- si : les sous-buts non encore résolus,

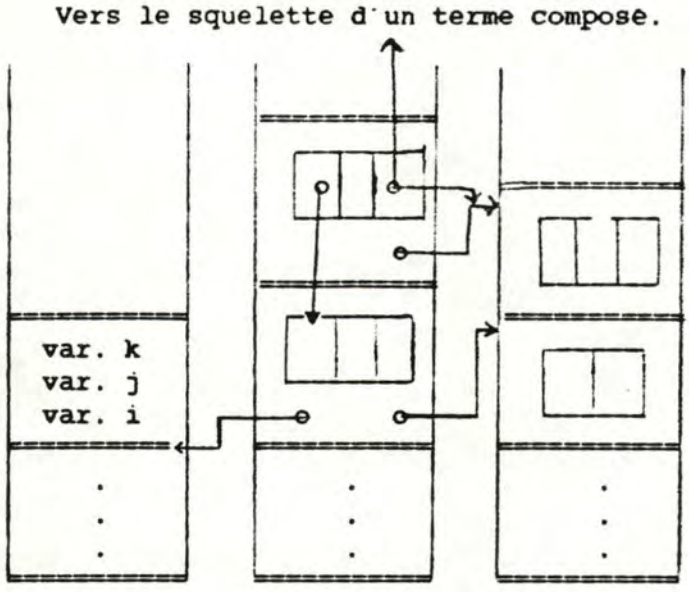

Figure 5.7.

Lorsque "s3" et "s4" seront exécutés, il faudra se souvenir que leur appel n'a pas été directement engendré par "b1" mais par un bloc intermédiaire. Par conséquent si "s3" ou "s4" devaient se révéler être l'opérateur "cut", ce n'est pas "b1" qui devrait être rendu déterministe mais la procédure intermédiaire qui l'était en fait déjà. Pour se faire, "b1" sera doté d'un moyen de se faire connaître comme n'étant pas le géniteur du "cut".

## 3. Mise en oeuvre d'un "garbage collector".

En conclusion, nous pouvons remarquer que les diverses implémentations mettent en oeuvre une organisation semblable de la zone de travail, à savoir trois piles, mises à jour à des moments identiques. La figure 5.8. résume ces organisations ainsi que les directions des références entre les diverses structures de données.

Toutes les références vers les piles globale ou de recopie trouvent leur origine dans la pile d'environnements. Par conséquent, recouvrer un bloc sur celle-ci peut, à certains moments, provoquer la disparition de la dernière référence à une donnée de l'autre pile.




Pile de : Trail. Travail .... Recopie.
Locale .... Globale.
(Environnements)

Figure 5.8.

Dès lors, lorsqu'un manque de place mémoire se fera sentir, l'usage d'un "garbage collector" sera requis pour récupérer les zones inaccessibles. La technique classique du "garbage collection" se déroule en deux temps. La première phase passe en revue toutes les zones utilisées les marquant au passage, tandis que la seconde collecte les locations non marquées, compactant les zones marquées. Le procédé est relativement coûteux en temps calcul, d'abord parce qu'il nécessite un double parcours de la zone à nettoyer ensuite parce que la compaction implique la nécessité de recalculer tous les pointeurs référencant des zones déplacées.

Exemple.

La figure 5.9. détaille un exemple de "garbage collection" en deux passes. Les zones accessibles sont d'abord repérées une à une et ensuite réorganisées séquentiellement, ce qui provoque une modification de l'adresse de certaines données. A la fin du processus, les zones inacessibles se trouvent rassemblées et sont réutilisables pour une allocation ultérieure.


Etat initial. -> Marquage. -> Compaction.

N.B. : Les zones accessibles sont hachurées.

Figure 5.9.

En Prolog, et dans le cas du partage de structure, nous dirons qu'un bloc global est actif si le bloc d'activation correspondant existe toujours. Les autres blocs seront passifs. Le "garbage collection" ne peut espérer une récupération que sur ces derniers qui correspondent à des procédures qui ont été achevées de manière déterministe. Les blocs actifs contiennent les variables qui, rappellons-le, peuvent encore être référencées au cours de la démonstration et donc ne peuvent être éliminées.

Le marquage démarrera des variables contenues dans les blocs locaux. La phase de compaction ne peut être systématique ainsi que le fait remarquer David Warren [WARR, 1977], car il se peut qu'une variable récupérable soit coincée entre deux autres qui ne le sont pas. Dans ce cas, puisqu'un environnement global est un vecteur dont les éléments sont accessibles suivant leur position relative, retirer la variable fausserait l'organisation de la structure de données. Par conséquent, seules les variables non marquées et se situant à l'une ou à l'autre extrémité du vecteur pourront être recouvrées.

En ce qui concerne la recopie intermédiaire, le principe est identique, sauf que cette fois, la phase de compaction ne subit aucune restriction.

Sur base de l'observation que les seules variables locales dont la valeur risque encore d'être référencée sont celles qui apparaissent au moins une fois dans la tête de la clause, Maurice Bruynooghe, [BRUY, 1984], propose quant à lui un algorithme optimisant la phase de marquage, espérant par la même occasion récupérer sensiblement plus d'espace que par la méthode classique. Dès lors, le lecteur perspicace pourrait s'interroger sur la raison pour laquelle les autres variables locales sont conservées sur la pile d'environnement. Maurice Bruynooghe argue que le gain d'espace serait marginal en regard de la

complexité de l'opération de récupération..

Cette dernière remarque nous permettra de conclure le chapitre en faisant remarquer que de part son coût en temps calcul, le "garbage collection" ne devrait être considéré que comme une opération de la dernière chance. Les deux premières techniques sont en effet nettement plus efficaces et permettent déjà des gains d'espace mémoire très conséquents.

# Chapitre 6: Les prédicats évaluables.

Les prédicats évaluables sont fournis dans un système Prolog pour deux raisons distinctes :

- Ce sont des fonctions indispensables dont la définition ne peut être obtenue en Prolog pur. Tel est le cas, par exemple du calcul arithmétique ou des opérations d'entrée et de sortie.
- Ce peut aussi être des procédures d'usage très fréquent ou pratique dont l'implémentation est relativement peu coûteuse pour l'interpréteur mais fournit un outil très appréciable au programmeur.

Ces prédicats sont le plus souvent écrits en langage machine et il n'est pas rare que leurs arguments soient typés. En outre, leur exécution peut donner lieu à des effets de bord, c'est à dire que leur action sur les structures de travail de l'interpréteur dépasse le cadre de simples instanciations de leurs arguments.

L'exécution de ces prédicats ne correspond plus à une résolution telle que nous l'avons vu précédemment; lorsque l'interpréteur détecte leur appel, il génère un branchement à une procédure écrite en assembleur ou dans un langage de programmation de type classique. Les paramètres sont le plus souvent passés par référence, ce qui d'une part explique la présence des effets de bord et d'autre part s'avère plus efficace qu'un passage par valeur.

La détection par l'interpréteur des appels de ces procédures est assez aisée. Celui-ci vérifie que tout prédicat pour lequel n'existe aucun paquet dans le programme n'est pas repris dans une liste des prédicats évaluables, auquel cas un branchement vers la routine appropriée est exécuté en lieu et place du rapport d'échec auquel on serait normalement en droit de s'attendre.

La liste que nous proposons ici est loin d'être exhaustive mais elle en reprend cependant les principaux, à savoir les opérations d'entrée et de sortie, quelques prédicats offrant un contrôle sur l'interpréteur, dont le célèbre "slash", les prédicats arithmétiques et ceux de gestion dynamique du code source d'un programme.

## 1. Les opérations d'entrée et de sortie.

Jusqu'à présent, nous n'avons laissé entrevoir qu'une seule possibilité d'échange d'information entre un programme Prolog et son utilisateur, à savoir poser des questions paramétrées auxquelles l'interpréteur s'efforcait de répondre par le biais de substitutions. Cependant, dans de nombreux cas, il peut être intéressant de disposer d'une interaction plus poussée. C'est la raison de l'introduction de

prédicats réalisant les opérations d'entrée et de sortie.

Les systèmes Prolog reconnaissent une unité courante en écriture ainsi qu'une autre en lecture. Par défaut, ces unités sont respectivement l'écran et le clavier. Cependant, pour permettre l'accès à des fichiers sur d'autres supports, deux prédicats permettent de modifier ces unités. Chaque fichier est ainsi considéré comme une unité particulière avec laquelle le programme a la possibilité de converser.

Ces deux prédicats se spécifient de la manière suivante :

Unitésortie(X) : lors d'un appel, ce prédicat réussit lorsque la variable X est instanciée au nom de l'unité à utiliser couramment en sortie. Si X n'est pas instanciée lors de l'appel, le prédicat réussit en lui associant la valeur de l'unité couramment utilisée.

Unitéentrée(X) : lors d'un appel, ce prédicat réussit lorsque la variable X est instanciée au nom de l'unité à utiliser couramment en entrée. Si X n'est pas instanciée lors de l'appel, le prédicat réussit en lui associant la valeur de l'unité couramment utilisée.

Les entrées et sorties effectives sont réalisées au moyen des quatre prédicats suivants :

incar(X) : instancie la variable X à la valeur d'un caractère lu sur l'unité d'entrée.

outcar(X) : écrit sur l'unité courante de sortie la valeur du caractère associé à X. Si X n'est pas instancié à la valeur d'un caractère, le prédicat échoue.

interm(X) : instancie X à la valeur d'un terme lu sur l'unité courante d'entrée.

outterm(X) : écrit sur l'unité courante de sortie la valeur du terme à laquelle est instanciée la variable X. Si la variable n'est pas instanciée, le prédicat échoue.

En outre, comme en Prolog les fichiers sont principalement utilisés pour stocker des programmes, une primitive particulière, consult(X), est ajoutée pour permettre la lecture d'un ensemble de clauses sur le fichier dont le nom est instancié à X et les ajouter au programme déjà chargé en mémoire. Notons que certaines implémentations permettent que la variable X soit instanciée à une liste de noms de fichiers.

D'autre part, le prédicat save(X) permet de sauver l'état courant du programme en cours d'exécution, c'est à dire en ce compris l'interpréteur et ses piles, sur le fichier dont le nom est associé à X.

Enfin, l'effet de tous ces prédicats n'est jamais remis en question par un retour arrière. Par conséquent, les instanciations comme les modifications au programme et aux fichiers dont ils sont responsables ne sont pas défaites.

## 2. Le contrôle de l'interpréteur.

### 2.1. Le "slash" ou "cut".

"Slash", noté "/" ou "!", est le prédicat évaluable le plus connu de Prolog. Il s'utilise pour restreindre le nombre d'alternatives permettant d'effacer un but d'où son surnom de "cut" puisqu'il élague certaines branches "ou" de l'espace de recherche. [WINS, 1984].

L'effacement du "cut" réussit toujours et possède une double conséquence. D'abord, il rend déterministe la procédure au sein de laquelle il est exécuté : si la clause qui le contient échoue ultérieurement à cet effacement, aucune autre règle du paquet ne sera essayée. En outre, il agit comme une barrière au sein de la règle dans laquelle il apparaît fixant définitivement les choix posés pour l'effacement des buts qui lui sont antérieurs dans le corps de la clause. En cas de retour arrière au-delà du "cut", la clause échoue donc d'office.

Exemple.

Soit la procédure :

```
p :- a , b / c , d.
p :- e , f.
p :- g.
```

Lors de son appel, la première clause sera d'abord essayée. Supposons que les buts "a" et "b" aient été prouvés, le "cut" est alors exécuté. A partir de là, le système doit absolument prouver la conjonction "c , d", en utilisant sa technique habituelle, pour assurer le succès de la procédure "p". En aucun cas, les choix de clauses ayant permis d'effacer les littéraux "a" et "b" ne seront remis en question, et en aucun cas, les deux autres clauses du paquet ne seront envisagées. Ainsi, si la conjonction

"c ^ d" devait échouer, la procédure "p" échouerait avec elle.

Le prédicat "cut" a une double utilité pratique :

- Il peut indiquer au système qu'il a trouvé, dans cette situation particulière, une règle idéale ou encore la seule règle possible pour résoudre un but particulier.

  Cet usage permet d'améliorer la rapidité de l'interpréteur en lui signalant qu'il est inutile de chercher des solutions alternatives soit qu'elles soient moins bonnes ou superflues, soit qu'elles n'existent pas.

- Il permet en combinaison avec le prédicat "fail" dont le rôle est d'échouer systématiquement, d'indiquer au système que s'il est parvenu à un point donné de la démonstration d'un sous-but, il doit abandonner la résolution de la clause qui a engendré ce dernier sous-but.

Exemple.

Suivant le programme :

```
a :- b1 ^ b2 ^ b3.
a :- b4 ^ b5.
b2 :- c1 / fail.
b2 :- c3 ^ c4
```

le "/ fail" apparaissant dans la première clause du paquet "b2" a pour but de forcer l'interpréteur à examiner la seconde définition de la procédure "a" lorsque "c1" peut être prouvé.

Ce dernier usage est également à la base de l'implémentation de la négation sur les systèmes où celle-ci n'est pas fournie en tant que prédicat évaluable.

```
not(P) :- P / fail.
not(P) :-.
```

Le prédicat "not" échoue si le but P réussit ou vice versa. Cette définition porte d'ailleurs le nom de négation par l'échec. Elle se base sur l'hypothèse du monde fermé signifiant que tout ce que le système ne peut prouver est d'office présumé faux.

L'implémentation concrète du "cut" revient à supprimer les blocs de travail correspondant aux appels précédents dans la clause en cours de résolution et à éliminer le point de choix éventuellement enregistré pour la clause dont le corps contient le "cut".

## 2.2. Le prédicat "geler".

L'hypothèse du monde fermé pose par ailleurs souvent problème en Prolog. En effet, toute réponse négative peut résulter soit d'un échec dans la démonstration, soit d'un manque d'information ne permettant pas de mener celle-ci à son terme, auquel cas la réponse est plutôt "je ne sais pas" que "faux". Dans la pratique, cette situation peut se révéler gênante : l'échec aurait souvent pu être évité si la décision avait été reportée, c'est à dire si l'instanciation de l'une ou l'autre variable avait été attendue pour appliquer une certaine procédure.

Le problème se résoud en Prolog par une programmation lourde et généralement au prix de la clarté du programme. C'est la raison pour laquelle le Prolog II de Marseille fournit le prédicat "geler".

L'appel "geler(V , T)", où V sera une variable et T un but à résoudre s'efface toujours avec succès, mais deux cas se présentent :

- si V est libre, l'effacement de T est retardé jusqu'à l'instanciation de V. Dans la pratique, l'interpréteur résoud le sous-but suivant, avant de réessayer à nouveau le but "geler (V , T)".

- si V est déjà instanciée, le but T est effacé normalement.

Il reste évidemment du ressort du programmeur d'assurer l'instanciation de la variable gelée à un moment ou à un autre.

## 2.3. Quelques autres prédicats de contrôle.

Nous reprendrons dans cette section une série de prédicats généralement utiles pour contrôler le déroulement d'un programme tels comparaison des deux structures, test d'état des variables, test de types d'arguments. Leur implémentation est très aisée et se résume la plupart du temps à un simple test sur le contenu des piles de travail de l'interpréteur.

dif(T1 , T2) : assure que les termes T1 et T2 possèdent une structure différente. Dans le cas contraire, le prédicat échoue, provoquant un retour arrière qui devrait en principe déboucher sur une modification des instanciations de ces deux termes. "Dif" permet souvent d'éviter l'emploi du "cut" et donc de permettre la lecture déclarative du programme.

bound(X) : s'efface si la variable X est instanciée, échoue dans le cas contraire.

free(X) : est le pendant de "bound". Il ne réussit que si la variable X est libre. Si elle est instanciée, il échoue.

integer(X) : réussit si la valeur ultime liée à la variable X est un entier; échoue dans tout autre cas.

real(X) : réussit si la valeur ultime liée à la variable X est un réel; échoue sinon.

tuple(X) : réussit si la valeur ultime liée à la variable X est un terme composé; échoue autrement.

atom(X) : réussit si la valeur ultime liée à la variable X est une constante de type atomique; échoue sinon.

## 3. Les opérations arithmétiques.

Toutes les opérations arithmétiques sont réalisées par le biais de la règle prédéfinie val(X , Y) dont le but est d'évaluer des expressions notées sous forme fonctionnelle :

expression_arithmétique := entier ou opérateur(argument , argument).
argument := expression_arithmétique.

Les opérateurs qui peuvent être utilisés pour construire l'expression arithmétique sont les quatre opérateurs classiques, plus (+), moins (-), fois (*) et divisé par (/) ainsi que les opérateurs de division modulaire (div et mod). Les opérateurs de comparaison (=, <>, <, >, =< et >=) sont également disponibles. La valeur "vrai" correspond alors à l'entier 1 et "faux" à 0.

Le succès du prédicat est subordonné à l'instanciation préalable de Y à une structure qui puisse s'interpréter sous forme d'une expression arithmétique, ceci pour un simple raison d'efficacité : le processeur est alors réservé au calcul arithmétique pur, ceci aux dépens du calcul inférentiel. Cette structure est en effet évaluée pour donner un entier appelé résultat. En Prolog, le calcul arithmétique se limite généralement aux entiers. Remarquons toutefois que le Prolog II_V2 de Marseille étend ce calcul aux réels.

L'évaluation est classique : les arguments sont évalués récursivement et doivent eux-mêmes donner des entiers, ensuite l'opérateur leur est appliqué pour fournir le résultat. Toute erreur au cours de l'évaluation est sanctionnée par un échec du prédicat

"val".

Enfin dans un troisième temps, le rèsultat est unifiè à X; la primitive "val" rèussit ou èchoue dès lors en fonction du verdict de cette unification.

## 4. La gestion du code source.

Nous dèsignons principalement sous le vocable de primitives de gestion du code source les deux prèdicats èvaluables permettant l'ajout et le retrait de clauses dans un programme.

assert(Y , X) : ajoute la clause instanciée à la variable X dans le paquet concernè à une position prècisèe par un entier associè à Y. Si la structure associèe à X n'est pas une clause, le prèdicat èchoue. Si Y est laissèe libre lors d'un appel, la clause est en gènèral placée à la fin du paquet et la variable instanciée à son numèro d'ordre.

retract(X) : permet de supprimer la première clause du programme s'unifiant à une clause constituant l'instanciation de X. Si la structure associèe à X n'est pas une clause ou si aucune clause du programme n'y correspond, le prèdicat èchoue.

Il est très important de noter qu'en cas de retour arrière sur un appel de "assert" ou de "retract", l'effet antèrieur n'est pas supprimè.

Exemple.

En consèquence, tout un paquet pourrait se voir èliminè par le simple jeu des retours arrière. Dans le cas du programme suivant :

```
p(X) :- q(X) , r(Y).
p(X) :- s(X).
p(X) :- t(V).
```

le but :

```
:- retract( p(X) :- A. ) , fail.
```

où " p(X) :- A. " est une clause dont le corps inconnu est symbolisè par une variable, aura pour effet d'effacer toutes les clauses du programme tout en signifiant un èchec à l'utilisateur.

Dans toutes les versions de Prolog, il existe d'autres primitives de gestion du programme. Nous ne les reprendrons pas ici car elles sont très spécifiques et dépendent fortement de l'organisation du code source de chaque implémentation.

Enfin, en guise de conclusion à ce chapitre, nous rappelerons que la liste ci-dessus est loin d'être exhaustive; elle se limite aux principales primitives offertes par chaque système Prolog. Pour être tout à fait complet, nous signalerons l'existence de prédicats permettant des opérations sur des chaînes de caractères, la manipulation de structures complexes telles les listes ou encore la gestion du temps (Cfr. Prolog II-V2 de Marseille. [GIAN et alt., 1985]).

# Chapitre 7: Voies d'améliorations de l'interpréteur.

Le grief principal fait à la programmation en logique reste encore actuellement l'efficacité de ses interpréteurs : il sont très lents comparativement à d'autres langages. C'est pourquoi de nombreuses recherches ont été effectuées dans un but d'amélioration. Ce chapitre tente de résumer ces travaux.

Nous débuterons par un recensement des trop rares outils fournis par l'environnement de programmation.

L'amélioration la plus tangible est certainement le système de David Warren, conçu à l'origine pour les processeurs DEC-10 de Digital et aujourd'hui adapté à d'autres machines, offrant outre la possibilité de compiler les clauses d'un programme un système d'indexation permettant d'écarter systématiquement les clauses inutiles dans certains cas particuliers.

Dans un troisième temps, nous nous attarderons sur le point focal de la recherche actuelle : l'introduction du parallélisme. Nous étudierons plus particulièrement le Concurrent Prolog de Shapiro qui présente le double avantage de constituer un système concret et d'être le langage de base du projet japonais d'ordinateurs de cinquième génération.

Enfin pour conclure ce dernier chapitre, nous examinerons l'état de la recherche dans le cadre de la mise au point d'un retour arrière intelligent.

## 1. Nécessité d'un bon environnement de programmation.

La plupart des difficultés rencontrées par les utilisateurs du langage Prolog tiennent moins aux performances de l'interpréteur qu'à la carence en un bon environnement de programmation. Nous regrouperons sous ce vocable tous les outils d'aide au développement de programmes tels éditeurs de clauses et systèmes de mise au point, car quoique prétendent les inconditionnels de la programmation en logique, les erreurs existent toujours en Prolog.

Prolog devrait aussi permettre l'accès de l'utilisateur au système d'exploitation résidant, ainsi qu'une communication avec d'autres logiciels de la machine hôte. La résolution de ces problèmes sera sans doute facilitée par l'apparition de machines dont le langage de base sera Prolog lui-même.

Pour notre part, nous nous bornerons à énumérer les solutions actuelles, renvoyant le lecteur intéressé aux travaux de Marc Tizzano, [TIZZ, à p.], sur le développement d'environnements de programmation pour le langage Prolog.

## 1.1. L'édition des programmes.

Les éditeurs incorporés, souvent écrits eux-mêmes en Prolog, sont généralement très limités et peu pratiques d'usage. Leurs principales fonctions se résument à gérer un pointeur de clause courante, à éditer, supprimer la règle courante ou encore à insérer une nouvelle règle dans le programme.

Cependant, Prolog permet également l'accès à des éditeurs extérieurs. L'utilisateur dispose alors d'outils familiers et généralement très sophistiqués.

## 1.2. La mise au point des programmes.

Deux types de traces sont possibles en Prolog. La première est assez sommaire : elle imprime tous les effacements de buts qui ont réussi. Hélas, son principal défaut est d'être tellement volumineuse qu'elle en devient rapidemment inutilisable. Toutefois, elle permet à l'utilisateur de suivre le déroulement de son programme au pas à pas.

Un second moyen consiste à faire imprimer la valeur des arguments d'un appel avant son effacement. Prolog permet son implémentation d'une manière relativement rapide. Il suffit en effet d'une règle dont la fonction est d'ajouter en tête d'une procédure une clause assurant l'impression de l'appel.

```
trace(U) :- assert(1 , U :- outterm(U)
                          , outcar(CR/LF)
                          , fail. ).
```

Pour le reste, il y a les techniques classiques de l'informatique , c'est à dire relire et tester le programme partie par partie, ce qui est relativement facilité par la modularité des procédures Prolog, ainsi que l'impression de valeurs particulières au cours de la résolution.

## 2. Le DEC-10 Prolog de David Warren.

L'implémentation que propose David Warren pour les processeurs DEC-10 de Digital, [WARR, 1977] est assez remarquable car elle se distingue des systèmes antérieurs par deux innovations :

- la possibilité de compiler les clauses Prolog en instructions assembleur, ce qui accroit considérablement la vitesse d'exécution des programmes.
- un mode particulier d'indexation des clauses au sein des procédures permettant d'éviter les essais voués à coup sûr à l'échec.

### 2.1. La compilation des clauses.

La compilation des clauses est certainement l'innovation la plus importante dans l'implémentation de David Warren. La méthode se base sur le fait qu'une exécution se résume essentiellement à une séquence d'unifications visant, comme nous le savons, à mettre en correspondance deux termes à savoir un sous-but à effacer et la tête d'une clause du programme, ceci par le biais d'instanciations des variables y apparaissant.

Le principe de la compilation revient à remplacer la mémorisation littérale de chaque clause d'un paquet par une séquence d'instructions réalisant en premier lieu l'unification de la tête de celle-ci et ensuite l'appel de chaque littéral du corps.

La caractéristique principale de ce procédé est que la procédure d'unification est remplacée le plus souvent par une série de simples instructions d'affectation. Son origine réside dans les quelques observations suivantes :

1. Dans toute tête de clause, la première occurrence de chaque variable, toujours libre au moment où débute son exemplarisation, peut directement être instanciée par le terme correspondant dans le sous-but à effacer.

2. En outre, et comme c'est souvent le cas, si cette variable n'a qu'une seule occurrence dans la tête de la clause, il n'est pas nécessaire de lui associer une location mémoire afin d'y placer sa valeur si elle est de type passe-partout. Aucune opération ne doit donc être effectuée : le gain est relativement considérable. Notons cependant que les variables temporaires ne seront mémorisées que jusqu'à achèvement des instructions d'unification. L'espace qui leur est reservé sera alors directement récupéré.
3. La génération du code pour les occurrences ultérieures de variables est un peu plus complexe et impliquera l'appel

d'une sous-routine d'unification. Il en va de même pour les termes constants.

4. Le code produit par les termes composés doit distinguer deux cas indiscernables à priori :
   - Le terme composé sera unifié à une variable libre du sous-but. Dans ce cas, une nouvelle molécule doit être construite et assignée à cette variable. Cette dernière sera en outre chargée sur le "trail" ce qui permettra de la libérer en cas de retour arrière.
   - Le terme composé doit être unifié à une structure différente d'une variable. Ce cas comprendra un test comparant les symboles fonctionnels principaux des deux structures ainsi que leur arité suivi des instructions produites par la compilation récursive de leurs arguments.

5. Enfin, le corps de la clause peut directement être remplacé par une séquence d'appels de procédures, chacune avec ses arguments respectifs.

Il est essentiel de noter que l'emploi de l'option de compilation empêche l'utilisation dynamique des prédicats "retract" et "assert".

Exemple.

Montrons sur l'exemple classique de concaténation de deux listes le type de résultat produit par la compilation.

```
concatenate (nil , LA , LA) :- .
concatenate (cons (X , LA) , LB , cons (X , LC)) :-
                                    concatenate (LA , LB , LC).


concatenate : initialiser un nouveau bloc local.
              utiliser la "clause1".
              si succès alors succès de concatenate.
              utiliser la "clause2".
              si succès alors succès de concatenate.
                        sinon échec de concatenate.

clause1  : déclarer LA temporaire.
           réserver une location pour LA.
           si pas d'unification entre "nil" et le premier
              argument de l'appel,
            alors retour avec échec de clause1.
            sinon
              LA := deuxième argument de l'appel.
              Si pas d'unification entre la valeur de LA
                 et le troisième argument de l'appel,
               alors retour avec échec de clause1.
               sinon
```

```
                    réclamer l'espace réservé à LA.
                    retour avec succès de clause1.

clause2  : déclarer LB locale.
           réserver une location pour LB.
           déclarer X, LA, LC globales.
           réserver une location pour X, LA, LC.
           représenter "cons(X , LA)".
           représenter "cons(X , LC)".
           { si le premier argument de l'appel est une
                variable libre
             alors
               construire une molécule représentant
                l'instanciation de "cons(X, LA)".
               X  := "libre".
               LA := "libre".
               lier la molécule au premier argument de
                l'appel.
               placer cet argument sur le "trail".
             sinon
              si pas d'unification entre "cons(X ,  LA)" et
                le premier argument de l'appel,
             alors retour avec échec de clause2.}
           LB := deuxième argument de l'appel.
           { si le troisième argument de l'appel est une
                variable libre
             alors
               construire une molécule représentant
                l'instanciation de "cons(X, LC)".
               X  := "libre".
               LC := "libre".
               lier la molécule au troisième argument de
                l'appel.
               placer cet argument sur le "trail".
             sinon
              si pas d'unification entre "cons(X , LC)" et
                le troisième argument de l'appel,
             alors retour avec échec de clause2.}
           appel ( concatenate(LA,LB,LC) ).
           si échec de l'appel précédent,
            alors retour avec échec de clause2.
            sinon retour avec succès de clause2.
```

David Warren propose de réaliser ces instructions qui ne sont naturellement pas toujours très simples au moyen de macros paramétrées.

## 2.2. Indexation des clauses.

Sauf usage explicite du "cut", le principe de base d'un système Prolog pour résoudre un appel est d'essayer en séquence toutes les clauses du paquet correspondant. Le plus souvent, c'est l'échec immédiat de l'unification qui provoque le rejet d'une clause à l'essai. Les performances d'une telle méthode se dégradent relativement vite pour les procédures comprenant un très grand nombre de clauses dont une faible proportion seulement s'applique à un appel particulier.

C'est typiquement le cas de clauses dont de nombreux arguments du prédicat de tête ne sont pas des variables. Il s'agit alors souvent de tableaux de données plutôt que de procédures.

Exemple

Le prédicat "population" défini ci-dessous qui procure pour chaque pays sa population en millions d'habitants en est un exemple.

```
population(Chine , 825) :-.
population(Inde , 586) :-.
population(Urss , 252) :-.
population(Usa , 212) :-.
population(Belgique , 10) :-.
```

En pratique, l'idéal serait que l'accès à de telles clauses soit possible sur un index plus large que simplement le prédicat de tête. La solution adoptée par David Warren est assez simple et consiste, dans la mesure du possible, à indexer les clauses par le prédicat de tête et le symbole fonctionnel du premier argument de celui-ci.

Les clauses d'une procédure sont, lors de la compilation séparées grâce à des instructions particulières en deux groupes : celles dont le premier argument du prédicat de tête est une variable et les autres. Le premier groupe reçoit le nom de section générale et le second de section spéciale. Lorsqu'un appel utilise les clauses, celles-ci sont essayées une à une de la manière habituelle tant qu'elles font partie de la section générale. Au contraire, lorsque des clauses de la section spéciale doivent être utilisées, si le premier argument de l'appel permet d'exploiter le rangement indexé (ce qui n'est pas le cas si lui-même est une variable libre), alors quelques instructions permettent d'essayer directement la première clause qui a quelques chances de réussite, à savoir celle dont la clé d'index correspond à celle fournie par l'appel ou la première

clause suivante appartenant à la section générale.

Le double choix que représente le mode d'indexation, à savoir sur le premier argument du prédicat et seulement celui-là n'est pas purement arbitraire : il résulte de l'absolue nécessité d'efficacité de ce système. Le fait que ce soit le premier argument qui est choisi comme complément de clé est assez arbitraire mais s'explique par sa simplicité et sa facilité d'exploitation. D'autre part, si David Warren a décidé de se limiter à un seul argument, c'est pour éviter la génération d'un trop-plein d'information d'indexation qui n'aurait pour résultat pratique que le ralentissement de l'interpréteur. Le gain obtenu d'un côté se perdrait alors d'un autre. Cette décision ne constitue pas en soi une restriction pour le programmeur qui peut, si besoin est, aisément réécrire le programme pour bénéficier à plein des avantages de la technique d'indexation. [WARR, 1977].

Exemple

A titre d'illustration, considérons le procédure suivante :

```
p(X , Y) :- q(X).
p(X , Y) :- q(Y).
p(a , X) :- r(a).
p(b , c) :- r(c).
p(b , c) :- s(b).
p(X , Y) :- p(a).
```

Celle-ci prendra la forme compilée :

```
p : initialiser un nouveau bloc local.
    noter point d'entrée section générale.
    utiliser la "clause1".
    si succès alors succès de p.
    utiliser la "clause2".
    si succès alors succès de p.
    si clé d'index utilisable,
     alors
       noter point d'entrée section spéciale.
       calculer clé d'index sur l'appel.
       si clé valide pour cette partie de
          section spéciale,
        alors
          utiliser la "clause[index]".
          si succès alors succès de p.
        sinon
          brancher à label.
     sinon
      utiliser la "clause3".
      si succès alors succès de p.
      utiliser la "clause4".
      si succès alors succès de p.
      utiliser la "clause5".
```

```
            si succès alors succès de p.
            annuler instruction suivante.
label :     noter point d'entrée section générale.
            utiliser la "clause6".
            si succès alors succès de p.
            sinon échec de p.
```

où "clausei", i = 1..6, correspondent aux adresses où ont été chargées les compilations de chacune des clauses de la procédure.

Dans le cas de l'appel ":- p(Z, T).", toutes les clauses seront essayées si Z est une variable libre tandis que dans le cas de l'appel ":- p(a, t)." les clauses quatre et cinq seront laissées de côté. De même, pour l'appel ":- p(f , X)" la sixième clause sera essayée immédiatement après la deuxième.

## 3. Le parallélisme en Prolog.

L'exploitation du parallélisme est certainement un des points centraux de recherche actuelle en programmation logique. La raison en est double. D'abord, la baisse des coûts du matériel laisse présager l'arrivée très prochaine sur le marché d'architectures multiprocesseurs. D'autre part, un langage comme Prolog offre de nombreuses opportunités d'introduction du parallélisme.

### 3.1. Possibilités de parallélisme en Prolog.

Les deux principales ouvertures du parallélisme d'un langage de programmation logique proviennent de la double liberté de choix laissée par le procédé de résolution linéaire sur les clauses de Horn.

parallélisme "ET" : L'avancée peut adopter n'importe quel ordre pour tenter de prouver les différents littéraux d'une conjonction, à condition de les sélectionner tous. Leur preuve peut également se pratiquer en parallèle.

parallélisme "OU" : De même, lorsque plusieurs clauses sont des candidates potentielles à l'effacement d'un but, il est tout à fait possible de les essayer toutes en parallèle et d'accepter la première solution ainsi obtenue.

Cette seconde possibilité n'est pas sans conséquences. En premier lieu, cela revient à implémenter une recherche en largeur d'abord en lieu et place de la recherche en profondeur d'abord et de gauche à droite. Donc, l'ordre des clauses dans un programme perd son importance et puisque les clauses sont utilisées en parallèle, il n'y a plus aucun besoin d'implémenter le retour arrière.

Les problèmes techniques concernant la synchronisation et la communication peuvent généralement se résoudre sans trop de difficultés. Nous verrons plus loin la solution adoptée par Ehud Shapiro pour son système.

Une troisième possibilité de parallélisme concerne la procédure d'unification. Classiquement, l'unification des termes composés consiste en un double test d'identité des symboles fonctionnels et de leur arité suivi par quelques appels récursifs visant à unifier les arguments entre eux. Sur une architecture parallèle, tous ces appels peuvent être menés simultanément, avec l'avantage qu'un éventuel échec est souvent détecté beaucoup plus rapidemment.

Beaucoup de problèmes résolus en Prolog se résument à une double phase de génération puis de test de solutions en coroutinage. Le parallélisme permet d'améliorer l'efficacité du comportement de l'interpréteur pour de tels cas. En effet, tout test d'une solution peut s'accomplir pendant la production de la suivante. Dans d'autres cas, la première partie d'une solution peut être vérifiée tandis la suite est construite. Cette alternative permet d'éviter la construction intégrale de solutions qui d'entrée de jeu ne répondent pas à la question.

Enfin, il est également possible de combiner l'utilisation des clauses en chaînage avant et arrière simultanément. La mise en oeuvre de cette éventualité est cependant très complexe.

## 3.2. Concurrent Prolog de Ehud Shapiro.

Le Concurrent Prolog de Shapiro, [SHAP, 1983], met en oeuvre les deux premières formes de parallélisme et résoud les problèmes de communication au moyen de variables partagées par plusieurs buts, tandis que la synchronisation est assurée en empêchant certains processus d'unifier des variables partagées à des termes constants ou composés. Cette dernière propriété est précisée par le programmeur en suffixant la variable par un point d'interrogation ("?"). Celles-ci porteront alors le nom de variables "read-only".

L'implémentation de l'annotation "?" consiste en un symbole fonctionnel d'arité un et en une extension à l'algorithme de John Robinson [ROBI, 1965] qui se définit de la manière suivante :

soient { X? , T } les termes à unifier,

Si T est une variable,
- alors : X? et T s'unifient par le biais de la substitution : { (X? , T) }.
- sinon : si X n'est pas une variable,
  - alors : unifier récurssivement X et T.
  - sinon : l'unification échoue.

Cette définition implique que le succès de l'unification dépend du temps. En effet, toute clause ayant échoué peut être réessayée ultérieurement et peut-être réussir alors.

Concurrent Prolog présente une seconde particularité syntaxique par rapport au Prolog standard : l'opérateur "commit", noté "|". Toute clause de Concurrent Prolog prend de la sorte la forme :

C :- G1 ^ G2 ^ ... ^ Gm | B1 ^ B2 ^ ... ^ Bn.
avec n,m >= 0.

où les "Gi" et les "Bj" (i = 1..m , j = 1..n) sont des littéraux. La conjonction des "Gi" (i = 1..m) se nomme la garde de la clause et celle des "Bi" (i = 1..n) son corps. Lorsque la garde est absente, le "commit" est omis mais reste virtuellement présent.

La signification déclarative du "commit" est simple : il agit comme le connecteur "^". Procéduralement, une telle clause s'interprète de la sorte : si un sous-but "A" s'unifie avec la tête "C" de la clause, le sous-but peut être effacé au moyen du corps de la clause si et seulement si la garde s'est au préalable terminée avec succès. En outre, l'exécution d'un "commit" élimine toute alternative à la clause qui le contient. Cette caractéristique rejoint en partie l'effet du "cut" décrit au chapitre précédent.

Dans la pratique, l'interpréteur de Concurrent Prolog s'axe autour de trois types de processus : le "and-dispatcher", le "or-dispatcher" et l'unifieur.

L'exécution débute avec une résolvante qui est passée au "and-dispatcher" et se poursuit ensuite de la sorte :

- Un "and-dispatcher" lorsqu'il reçoit une résolvante passe chaque sous-but de celle-ci à un "or-dispatcher" et attend que chacun de ceux-ci se termine et lui rapporte un succès. Lorsque tel est le cas, lui-même s'arrête avec succès.

- Un "or-dispatcher", lorsqu'il reçoit un sous-but à effacer, invoque pour chaque clause du programme pouvant éventuellement convenir, un unifieur de cette clause et du littéral correspondant. Le "dispatcher" attend alors que l'un des unifieurs signale un succès et lorsque que ce succès arrive, il s'arrête en rapportant lui-même un succès.

- Chaque unifieur tente en premier lieu de mener à bien son unification. Pour ce faire, il mémorise les instanciations qu'il fait sur les variables non "read-only" dans un espace réservé. Ceci assure la sécurité de chacun des unifieurs travaillant en parallèle sur un sous-but donné. Lorsque cette unification se termine avec succès, l'unifieur invoque un "and-dispatcher" avec la conjonction que forme la garde de la clause. Enfin, lorsque ce "dispatcher" lui aura signalé son succès, l'unifieur tentera d'assurer le "commit". Pour ce faire, l'unifieur doit en premier lieu obtenir du "or-dispatcher" la permission de le faire. Ceci fait, il tentera d'unifier les copies des variables placées dans son espace réservé avec celles correspondant à la situation réelle. Cette opération vise à empêcher la modification de variables qui auraient déjà été instanciées. En cas de réussite de ce "commit", les frères de cet unifieur ne pourront en tenter un second, ce qui élimine ainsi les possibilités alternatives. L'unifieur, quant à lui, déclenche alors un "and-dispatcher" avec le corps de la clause.

## 4. Affinement du retour arrière.

Le retour arrière systématique implémenté dans un système Prolog peut s'avérer assez inefficace. En effet, après avoir constaté un échec, le système remet simplement en cause le dernier choix qu'il a effectué. Cependant, dans certains cas, cette solution ne permet pas d'éviter que le même échec ne se reproduise.

Exemple.

Considérons l'exécution du programme suivant :

```
q(a) :-.
q(b) :-.
r(b) :-.
r(c) :-.
t(b) :-.
:- q(Z) ^ r(T) ^ t(Z).
```

Dans l'exemple ci-dessus, dont la figure 7.1. résume les résolutions, il est évident que la remise en question de la résolution du sous-but "r(T)" ne supprimait pas la cause de l'échec rencontré lors de l'effacement de l'autre sous-but "t(Z)". Une analyse un peu plus profonde permet de déterminer directement que la reprise doit se faire au niveau de la résolution du premier sous-but.

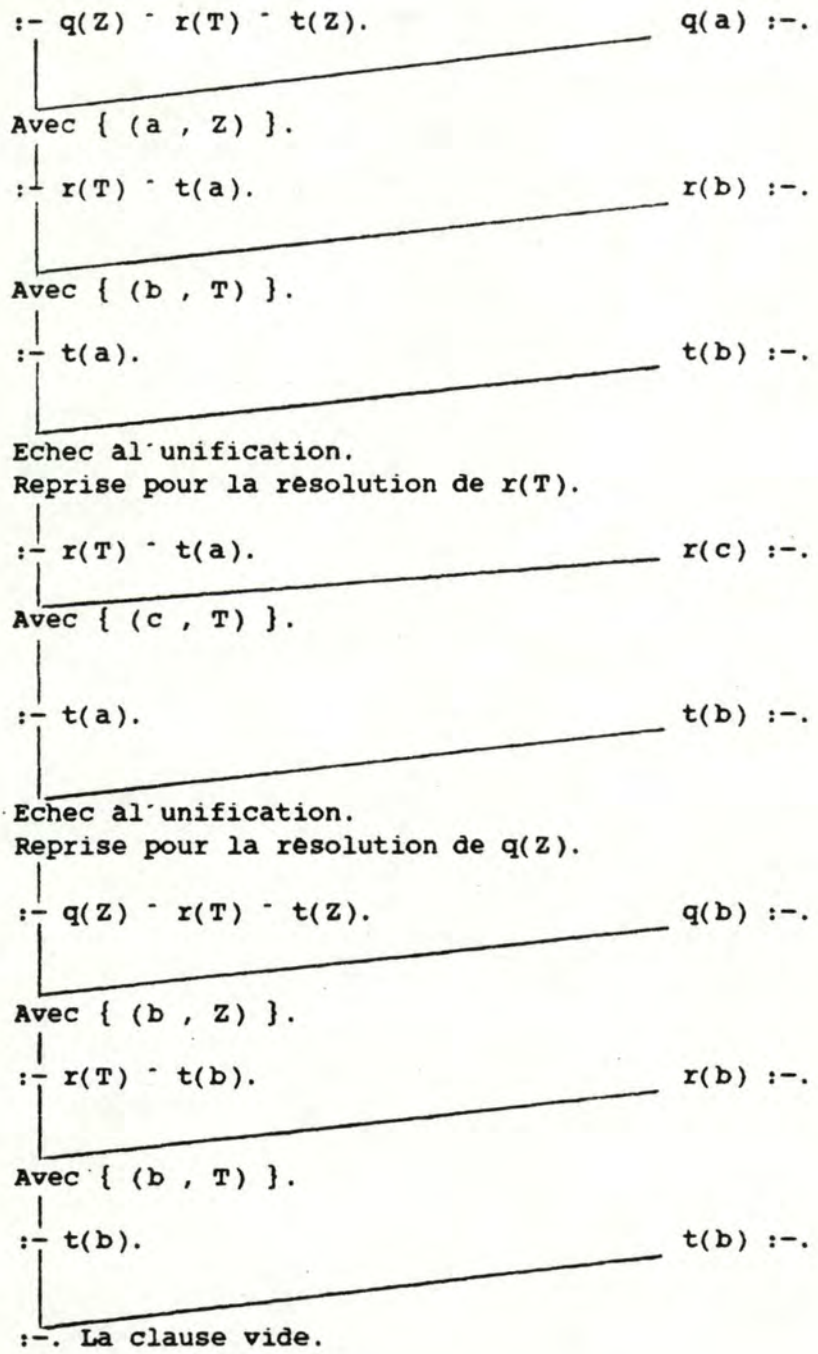



Figure 7.1.

Ce type de comportement peut aboutir à ralentir considérablement l'interpréteur. C'est la raison pour laquelle plusieurs chercheurs se sont penchés sur ce problème. En fait, celui-ci peut se résoudre de deux manières différentes :

- Au moyen de techniques de retour arrière intelligent, fondées sur d'imposantes théories, elles-mêmes développées par deux écoles à

savoir celle de Maurice Bruynooghe et de Luis Pereira, [BRUY et alt., 1984], et celle conduite par Philip Cox, [COX , 1984]. Dans la pratique, leurs travaux sont équivalents.

- La seconde méthode est également proposée par Philip Cox, un peu désabusé par le résultat de ses recherches sur la précédente et consiste simplement à fournir au programmeur des outils construits au sein du système lui permettant de produire des règles de sélection entre choix alternatifs au sein du programme même. La recherche dans ce domaine est ouverte.

La première méthode se base sur une analyse assez compliquée des unifications se déroulant lors de l'élaboration de l'arbre de preuve. Cette technique nécessite d'ailleurs une redéfinition de l'algorithme d'unification. En effet, alors que l'algorithme de Robinson, [ROBI, 1965], rapporte l'échec de l'unification dès qu'une incompatibilité est détectée, le retour arrière sélectif se doit de connaître toutes les causes de cet échec, ceci afin de pouvoir déterminer à coup sûr le choix de clause qui est à l'origine de cet échec.

Maurice Bruynooghe, [BRUY et alt., 1984], a tenté de mettre en application sa théorie. Sa conclusion est double : en premier lieu, la technique semble s'avérer profitable dans la pratique; d'autre part, il lui est nécessaire de simplifier sa théorie générale d'analyse car la mise en oeuvre de celle-ci dans son intégralité engendrerait dans la grande majorité des interpréteurs plus de problèmes qu'elle n'en résoudrait. Cependant, la méthode intégrale resterait avantageuse pour des interpréteurs très sophistiqués comme par exemple ceux utilisant les techniques de parallélisme.

[COX , 1984] et [BRUY et alt., 1984] présentent un résumé approfondi des travaux sur ce sujet. Nous y renvoyons le lecteur avide de plus de détails.

# Conclusion.

Nous avons donc passé en revue les grands principes de l'implémentation d'un interpréteur Prolog. A cet effet, nous avons consacré les deux premiers chapitres à quelques rappels théoriques indispensables à la compréhension de la suite de l'ouvrage. Ensuite, nous avons abordé l'étude des structures de données de l'interpréteur, du mode de représentation de la valeur d'une variable et des techniques de récupération de l'espace de travail inutile. Enfin, après avoir passé en revue une série de prédicats évaluables dont il est intéressant de doter tout système Prolog, nous nous sommes attardés à l'examen de quelques possibilités d'amélioration d'un tel système.

Au cours de cette étude, nous nous sommes efforcés de suivre la progression historique du développement des interpréteurs Prolog qui aboutit finalement à leur sophistication actuelle. Celle-ci nous semblait en effet découler d'une suite logique de nécessités successives et nous sommes certains que le lecteur qui réalisera son propre interpréteur la ressentira lui-même. Notre propre réalisation - succinte, nous le concédons volontiers - reprise à l'annexe deux en est un témoin.

Le langage Prolog est certainement promis à un bel avenir. Nous n'affirmerons pas qu'il éclipsera un jour ses concurrents, mais sa spécificité en fait toutefois un outil bien agréable face à ces derniers. Assurément, bien des améliorations devront encore être apportées à l'interpréteur avant de voir Prolog figurer dans la liste des plus répandus, mais qui oserait préjuger de l'avenir ?

Bibliographie.

[BATT et alt., 1973] : Battani G. et Meloni H.

Interpréteur du langage de programmation Prolog.
Rapport du Groupe d'Intelligence Artificielle. Université d'Aix-Marseille. 1973.

[BOIZ, 1985] : Boizumault Patrice.

Implémentation d'un interpréteur Prolog en Lisp.
Thèse de troisième cycle. Université de Paris VI. 1985.

[BOWE, 1979] : Bowen Kenneth A.

Prolog.
School of computer and information science.
Syracuse university. Sept. 1979.

[BOYE et alt., 1972] : Boyer R. S. et Moore J. S.

The sharing of structure in theorem proving programs.
dans : Machine intelligence 7. D. Michie and B. Meltzer editors.
Edimburgh University Press. 1972.

[BRUY, 1976] : Bruynooghe Maurice.

An interpreter for predicate logic programs. Part1 : basic principles.
Report CW10. Applied Mathematical and Programming Department.
Katolieke Universiteit Leuven. 1976.

[BRUY, 1982] : Bruynooghe Maurice.

The memory management of Prolog implementation.
dans : Logic Programming. K. Clark et S. Tärnlund editors. pp 83-98.
Academic Press. London. 1982.

[BRUY, 1984] : Bruynooghe Maurice.

Garbage collection in Prolog interpreters.
dans : Implementations of Prolog. J. A. Campbell editor. pp 259-267.
Ellis Horwood series in Artificial Intelligence. New-York. 1984.

[BRUY et alt., 1984] : Bruynooghe Maurice et Pereira Luis M.

Deduction revision by intelligent backtracking.
dans : Implementations of Prolog. J. A. Campbell editor. pp 194-215.
Ellis Horwood series in Artificial Intelligence. New-York. 1984.

[BURS et alt., 1977] : Burstall R. M. et Darlington John.

A transformation system for developing recursive programs.
dans : Journal of the ACM. Vol. 24, n. 1, January 1977. pp 44-67.

[CLAR et alt., 1977] : Clark Keith L. et Tärnlund Sten Ake.

A first-order theory of data and programs.
dans : Information processing 1977. B. Gilchrist editor. pp 939-944.
Elsevier North Holland Publishing Co. Amsterdam. 1977.

[CLOC et alt., 1984] : Clocksin William F. et Mellish Christopher S.

Programming in Prolog.
Springer Verlag. Berlin, Heidelberg. 1984.

[COLM, 1982] : Colmerauer Alain.

Prolog and infinite trees.
dans : Logic Programming. K. Clark et S. Tärnlund editors. pp 231-251.
Academic Press. London. 1982.

[COLM et alt., 1979] : Colmerauer Alain, Kanoui Henri et van Caneghem Michel.

Etude et réalisation d'un système Prolog.
Rapport 77030. Université d'Aix-Marseille. 1979.

[COLM et alt., 1983] : Colmerauer Alain, Kanoui Henri et van Caneghem Michel.

Prolog : bases théoriques et développements actuels.
dans : Techniques et sciences informatiques. Vol. 2, n. 2, 1983.
Academic Press. London. 1982.

[COX , 1984] : Cox Philip T.

Finding backtrack points for intelligent backtracking.
dans : Implementations of Prolog. J. A. Campbell editor. pp 216-233.
Ellis Horwood series in Artificial Intelligence. New-York. 1984.

[EMDE, 1982] : van Emden M. H.

An interpreting algorithm for Prolog programs.
dans : Procedings of the First International Logic Programming Conference. pp 56-64/bis. University of Marseille. 1982.
repris : Implementations of Prolog. J. A. Campbell editor. pp 85-110. Ellis Horwood series in Artificial Intelligence. New-York. 1984.

[EMDE et alt., 1976] : van Emden M. H. et Kowalski Robert A.

The semantics of predicate logic as a computer language.
dans : Journal of the ACM. Vol. 23, n. 4, october 1976. pp 733-742.

[ENNA et alt., 1984] : Ennals Richard, Briggs Jonathan et Brough Derek

What the naïve user wants from Prolog.
dans : Implementations of Prolog. J. A. Campbell editor. pp 376-386. Ellis Horwood series in Artificial Intelligence. New-York. 1984.

[FILG, 1984] : Filgueiras M.

A Prolog interpreter working with infinite terms.
dans : Implementations of Prolog. J. A. Campbell editor. pp 250-258. Ellis Horwood series in Artificial Intelligence. New-York. 1984.

[GALL, 1981] : Gallaire Hervé.

Le langage Prolog.
Support de cours d'ingéniorat en intelligence artificielle, reconnaissance des formes et robotique. Toulouse. 1981.

[GIAN et alt., 1985] : Giannesini Francis, Kanoui Henri, Pasero Robert et van Caneghem Michel.

Prolog.
Inter-éditions. Paris. 1985.

[GREU, 1977] : Greussay Patrick.

Contribution à la définition interprétative et à l'implémentation des lambda-langages. Thèse. Université de Paris VII. 1977.

[HARI et alt., 1984] : Haridi S. et Sahlin D.

Efficient implementation of unification of cyclic structures.
dans : Implementations of Prolog. J. A. Campbell editor. pp 234-249. Ellis Horwood series in Artificial Intelligence. New-York. 1984.

[KLUZ et alt., 1984] : Kluzniak F. et Szpakowicz S.

Prolog : a panacea ?
dans : Implementations of Prolog. J. A. Campbell editor. pp 71-84.
Ellis Horwood series in Artificial Intelligence. New-York. 1984.

[KOWA, 1979a] : Kowalski Robert A.

Algorithm = Logic + Control.
dans : Communications of the ACM. Vol. 22 n. 7, July 1979. pp 424-436.

[KOWA, 1979b] : Kowalski Robert A.

Logic for problem solving.
Elsevier North Holland Publishing Co. New-York. 1979.

[KOWA, 1982] : Kowalski Robert A.

Logic as a computer language.
dans : Logic Programming. K. Clark et S. Tärnlund editors. pp 3-16.
Academic Press. London. 1982.

[LECH, 1985] : Lecharlier Baudouin.

Reflexions sur le problème de la correction des programmes.
Thèse de doctorat. Facultès Notre Dame de la Paix. Namur. Jan. 1985.

[LISK et alt., 1975] : Liskov Barbara H. et Zilles Stephen N.

Specification techniques for data abstraction.
dans : IEEE Transactions on software engeenering. Vol SE-1, n. 1,
march 1975. pp 7-19.

[MART et alt., 1982] : Martelli Alberto et Montanari Ugo.

An efficient unification algorithm.
dans : ACM Transactions on Programming Languages. Vol. 4, n. 2, April
1982. pp 258-282.

[MELL, 1982] : Mellish Christopher S.

An alternative to structure sharing in the implementation of a Prolog
interpreter.
dans : Logic Programming. K. Clark et S. Tärnlund editors. pp 99-106.
Academic Press. London. 1982.

[MEYE, 1975] : Meyer Bertrand.

On formalism in specifications.
College of engeenering. Univ. of California, Santa Barbara. June 1975.

[NILS, 1971] : Nilsson Nils J.

Problem solving methods in artificial intelligence.
Mc Graw Hill. New York. 1971.

[ROBE, 1977] : Roberts Grant M.

An implementation of Prolog.
Mathematical Science Thesis. Department of Computer Science.
University of Waterloo, Canada. 1977.

[ROBI, 1965] : Robinson John Adam.

A machine oriented logic based on the resolution principle.
dans : Journal of the ACM. Vol. 12, n. 1, January 1965. pp 23-41.

[ROBI, 1968] : Robinson John Adam.

The generalized resolution principle.
dans : Machine intelligence 3. D Michie editor. pp 77-94.
Edimburgh University Press. 1968.

[ROBI, 1971] : Robinson John Adam.

Computational logic : the unification computation.
dans : Machine intelligence 6. B. Mitchie and H. Geltzen editors.
Edimburgh University Press. 1971.

[ROY , 1984] : Roy J.-P.

Lisp et Prolog en classe terminale.
Publication n. 53 Irem Paris VII. Paris. Septembre 1984.

[SHAP, 1983] : Shapiro Ehud Y.

A subset of Concurrent Prolog and its interpreter.
Technical report 003. Institute for new generation computer technology, (ICOT). Février 1983.

[SZER, 1977] : Szeredi P.

Prolog : a very high level language based on predicate logic.
Preprints of the second Hungarian Computer Science Conference.

[TARN, 1975] : Tärnlund Sten Ake.

Logic information processing.
University of Stockolm. 1975.

[TIZZ, à p.] : Tizzano Marc.

Environnements de programmation pour Prolog.
Projet de thèse. Université de Paris VI. A paraître.

[WARR, 1977] : Warren David H.

Implementing Prolog : compiling predicate logic.
D.A.I. research reports n. 39 and 40. University of Edimburgh. 1977.

[WARR, 1980] : Warren David H.

An improved Prolog implementation wich optimizes tail recursion.
Proceedings of the Logic Programming Workshop. Debrecen. 1980.

[WARR et alt., 1977] : Warren David H., Pereira Luis M. et Pereira Fernando

Prolog : the langage and its implementation compared with Lisp.
dans : Sigplan notices, Vol. 12, n. 8 (special issue), 1977. pp 109-115. (Sigart newsletter, n. 64).

[WINS, 1977] : Winston P. H.

Artificial intelligence. (Second edition).
Addison Wesley. Readings, Massachussetts. 1984.

# Annexe 1: Exemples de calcul d'un unificateur.

Dans cette annexe sont repris quelques exemples détaillés de calcul de l'unificateur le plus général de deux termes.

## 1. Premier exemple.

Soient les termes :

- t1 = f (X , h(X) , Y)
- t2 = f (g(Z) , W , Z)

Appliquons l'algorithme de Robinson pour en calculer l'unificateur le plus général.

### 1.1. Etape 1.

- S = {} ;

### 1.2. Etape 2.

- (t1)S et (t2)S ne sont pas identiques ;
- passer à l'étape 3 ;

### 1.3. Etape 3.

- k = 1 ;
- E = f(X , h(X) , Y) ;
- F = f(g(Z) , W , Z) ;
- E' = X ;
- F' = g(Z) ;

### 1.4. Etape 4.

- E' est bien une variable ;
- Passer à l'étape 5 ;

### 1.5. Etape 5.

- E' n'apparaît pas dans F' ;
- Passer à l'étape 6 ;

1.6. Etape 6.

- (t , X) = (g(Z) , X) ;

1.7. Etape 7.

- S = { (g(Z) , X) } ;
- Passer à l'étape 2 ;

1.8. Etape 2.

- (t1)S et (t2)S ne sont pas identiques ;
- Passer à l'étape 3 ;

1.9. Etape 3.

- k = 1 ;
- E = f(g(Z) , h(g(Z)) , Y) ;
- F = f(g(Z) , W , Z) ;
- E' = h(g(Z)) ;
- F' = W ;

1.10. Etape 4.

- F' est bien une variable ;
- Passer à l'étape 5 ;

1.11. Etape 5.

- F' n'apparait pas dans E' ;
- Passer à l'étape 6 ;

1.12. Etape 6.

- (t , X) = (h(g(Z)) , W) ;

1.13. Etape 7.

- S = { (g(Z) , X) , (h(g(Z)) , W) } ;
- Passer à l'étape 2 ;

1.14. Etape 2.

- (t1)S et (t2)S ne sont pas identiques ;
- Passer à l'étape 3 ;

1.15. Etape 3.

- k = 1 ;
- E = f(g(Z) , h(g(Z)) , Y) ;
- F = f(g(Z) , h(g(Z)) , Z) ;
- E' = Y ;
- F' = Z ;

1.16. Etape 4.

- E' est bien une variable ;
- Passer à l'étape 5 ;

1.17. Etape 5.

- E' n'apparaît pas dans F' ;
- Passer à l'étape 6 ;

1.18. Etape 6.

- (t , X) = (Z , Y) ;

1.19. Etape 7.

- S = { (g(Z) , X) , (h(g(Z)) , W) , (Z , Y) } ;
- Passer à l'étape 2 ;

1.20. Etape 2.

- (t1)S et (t2)S sont identiques ;
- Stop. Unification réussie.

2. Deuxième exemple.

Soient les termes :

- t3 = f (a , b)
- t4 = f (X , X)

Ceux-ci ne sont naturellement pas unifiables comme en témoigne l'algorithme de Robinson.

2.1. Etape 1.

-
- S = {} ;

2.2. Etape 2.

- (t3)S et (t4)S ne sont pas identiques ;
- passer à l'étape 3 ;

2.3. Etape 3.

- k = 1 ;
- E = f(a , b) ;
- F = f(X , X) ;
- E' = a ;
- F' = X ;

2.4. Etape 4.

- F' est bien une variable ;
- Passer à l'étape 5 ;

2.5. Etape 5.

- F' n'apparait pas dans E' ;
- Passer à l'étape 6 ;

2.6. Etape 6.

- (t , X) = (a , X) ;

2.7. Etape 7.

- S = { (a , X) } ;
- Passer à l'étape 2 ;

2.8. Etape 2.

- (t3)S et (t4)S ne sont pas identiques ;
- Passer à l'étape 3 ;

2.9. Etape 3.

- k = 1 ;
- E = f(a , b) ;
- F = f(a , a) ;
- E' = a ;
- F' = a ;

2.10. Etape 4.

- Ni E' ni F' ne sont des variables.
- Stop. Echec de l'unification.

3. Troisième exemple.

Enfin, les termes :

- t5 = f (X , Y)
- t6 = f (a , g(Y))

mettront en évidence l'utilité du test d'occurrence.

3.1. Etape 1.

- S = { } ;

3.2. Etape 2.

- (t5)S et (t6)S ne sont pas identiques ;
- passer à l'étape 3 ;

3.3. Etape 3.

- k = 1 ;
- E = f(X , Y) ;
- F = f(a , g(Y)) ;
- E' = X ;
- F' = a ;

3.4. Etape 4.

- E' est bien une variable ;
- Passer à l'étape 5 ;

3.5. Etape 5.

- E' n'apparaît pas dans F' ;
- Passer à l'étape 6 ;

3.6. Etape 6.

- (t , X) = (a , X) ;

3.7. Etape 7.

- S = { (a , X) } ;
- Passer à l'étape 2 ;

3.8. Etape 2.

- (t5)S et (t6)S ne sont pas identiques ;
- Passer à l'étape 3 ;

3.9. Etape 3.

- k = 1 ;
- E = f(a , Y) ;
- F = f(a , g(Y)) ;
- E' = Y ;
- F' = g(Y) ;

3.10. Etape 4.

- E' est bien une variable ;
- Passer à l'étape 5 ;

3.11. Etape 5.

- E' apparaît dans F' ;
- Stop. Echec de l'unification.

# Annexe 2: Implèmentation d'un interprèteur succint en Vlisp.

Cette annexe a pour but de montrer qu'il est relativement aisè de concrètiser les grands principes dècrits dans le prèsent ouvrage. Nous procèderons pour ce en trois temps. D'abord, nous mettrons en oeuvre le procèdè de rèsolution, ensuite, nous lui adjoindrons une implèmentation de l'algorithme d'unification et enfin nous introduirons un système permettant d'èlaguer certaines branches de l'arbre de recherche. Nous aboutirons ainsi à une maquette sans grandes prètentions mais ayant le mèrite d'être opèrationnelle d'un interprèteur Prolog.

Nous utiliserons le langage Vlisp pour programmer les diffèrents algorithmes.

## 1. La rèsolution.

### 1.1. Repràsentation du code source.

Dans ce premier temps, nous nous limiterons uniquement à des clauses de Horn sans variables. Chacune d'elles sera repràsentèe au moyen d'un liste dont le premier èlèment sera le consèquent et les autres formeront le corps. Ainsi les clauses

```
d :- a . b . c.
a :-.
```

s'ècriront

```
(d a b c)
(a)
```

Le programme sera lui-même une liste, valeur d'une variable globale que nous identifierons par *BC*. L'adjonction d'une clause à celui-ci se fera par le biais de la fonction "+" definie de la manière suivante :

```
(dmc + ()
     (let ((lu (read)))
          (ifn (listp lu)
               (print '(Une règle doit être une liste !))
               (setq *BC* (cons lu *BC*))
          )))
```

## 1.2. Construction du moteur d'inférences.

La résolution sera déclenchée par l'introduction au moyen de la fonction "-" d'une liste de buts qui s'interprétera comme un démenti.

```
(dmc - ()
     (let ((lu (read)))
          (ifn (listp lu)
               (print '(Un démenti doit être une liste !))
               (traiter (lu))
          )))
```

La fonction "traiter" est chargée d'effectuer la résolution :

```
(de traiter (liste_buts)
    (if (résoudre liste-buts)
        (print '(Oui. OK.))
        (print '(Non. Echec.))
    ))
```

La fonction "résoudre" s'assure que la clause vide est atteinte, et dans le cas contraire relance l'avancée au moyen de la fonction "démontrer" qui contrôle le déroulement de la preuve et les retours arrière.

```
(de résoudre (résolvante)
    (if (null résolvante)
        t
        (démontrer résolvante *BC*)
    ))

(de démontrer (liste_buts liste_axiomes)
    (cond
     ; toutes les clauses ont été essayées sans succès.
     ((null liste_axiomes) nil)
     ; le conséquent de la première clause correspond
     ; au premier but à démontrer.
     ((eq (car liste_buts) (caar liste_axiomes))
      (or
       ; la première clause est appliquée au premier sous-but.
       (résoudre (append (cdar liste_axiomes) (cdr liste_buts)))
       ; le choix n'était pas bon : il y a reprise.
       (self liste_buts (cdr liste_axiomes))
      ))
     ; la première clause du programme ne convenant pas, elle
     ; est rejetée. Le reste du programme est essayé.
     (t (self liste_buts (cdr liste_axiomes)))
    ))
```

## 1.3. Une première critique.

Cette version n'exploite pas la notion de procédure que nous avons mise en évidence dès le premier chapitre. Une manière très simple de corriger ce défaut consiste à placer chaque paquet sur la P-liste de l'atome représentant le prédicat, ceci sous un indicateur quelconque. La variable globale *BC* disparaît par la même occasion. Il suffit de redéfinir la fonction "+" :

```
(dmc + ()
     (let ((lu (read)))
          (ifn (listp lu)
               (print '(Une règle doit être une liste !))
               (put (car lu) 'règles
                    (cons lu (get (car lu) 'règles)))
          )))
```

et d'adapter les fonctions "résoudre" et "démontrer".

```
(de résoudre (résolvante)
    (if (null résolvante)
        t
        (démontrer résolvante (get (car résolvante) 'règles)))
    ))

(de démontrer (liste_buts liste_axiomes)
    (cond
     ; toutes les clauses ont été essayées sans succès.
     ((null liste_axiomes) nil)
     ; c'est la dernière clause à l'essai
     ((null (cdr liste_axiomes))
      (résoudre (append (cdar liste_axiomes) (cdr liste_buts)))
     ; il reste une clause donc création d'un point de reprise.
     (t (or (résoudre (append (cdar liste_axiomes)
                              (cdr liste_buts)))
            (self liste_buts (cdr liste_axiomes))
     ))))
```

## 2. Le principe d'unification.

Nous ne disposons pour l'instant que d'un système sans variables, ce qui, il faut bien en convenir n'est guère très puissant. Dans cette section, nous nous proposons d'introduire celles-ci et donc d'implémenter l'algorithme d'unification de Robinson.

## 2.1. Les structures de données.

Nous garderons la représentation d'une clause sous forme de liste. Cependant, puisque les littéraux pourront contenir des variables il devient impossible de les stocker sous forme d'atomes. Chaque littéral sera donc lui-même une liste dont le premier élément sera le symbole fonctionnel et le reste les arguments. Ceci nous impose de modifier très légèrement la fonction "+".

```
(dmc + ()
     (let ((lu (read)))
          (ifn (listp lu)
               (print '(Une règle doit être une liste !))
               (put (caar lu) 'règles
                    (cons lu (get (caar lu) 'règles)))
          )))
```

Les termes simples seront représentés sous forme atomique, les termes composés en syntaxe Vlisp, c'est à dire préfixée et parenthésée. Les variables feront partie d'un catalogue fixé au préalable par le biais de la fonction "setvar" et pourront être détectées au moyen du prédicat "var?".

```
(de setvar (lvar)
    (setq *lvar* lvar)
    (mapc lvar (lambda (x) (put x 'VAR t)))
)

(de var? (x)
    (and (litatom x) (get x 'VAR))
)
```

Les substitutions seront représentées par des A-listes, associant à chaque variable le terme qui lui est affecté par l'unification. Ce terme pourra éventuellement contenir des variables. Il s'agira donc d'une mise en pratique particulière du procédé de partage de structure. Rappelons que la création des valeurs des variables est dès lors très simple, mais que leur consultation demande l'exploration complète des chaînes de liaisons. Nous développerons dans ce but la fonction "appsub".

```
(de appsub (subst terme)
    (let ((terme terme)
          (cond
           ; Le terme est une variable.
           ((var? terme)
            (let ((aval? (assq terme subst)))
                 (if aval?
                      ; La variable a elle-même une valeur.
                      (appsub (subst (cdr aval?))
                      ; La variable est libre.
                      terme)))
```

```
        ; Le terme est une constante.
        ((atom terme) terme)
        ; Le terme est composé.
        (t (cons (self (car terme)) (self (cdr terme))))
    )))
```

A chaque instant, la valeur d'un terme peut maintenant être déterminée par l'intermédiaire de la fonction "appsub" à laquelle seront communiqués le squelette du terme et la substitution à y appliquer.

## 2.2. La fonction d'unification.

Nous pouvons maintenant construire une fonction d'unification. Celle-ci recevra en entrée deux termes à unifier ainsi que la pile d'environnements grâce à laquelle leurs instanciations pourront être reconstruites. Le résultat sera soit un message déclarant que les termes donnés ne sont pas unifiables, soit une nouvelle pile d'environnement, extension de la précédente, dans laquelle les deux termes seront identiques si nous leur appliquons la fonction "appsub".

```
(de unifier (e1 e2 env)
    (escape impossible
            ; Point de sortie immédiate pour un échec.
            (concilier (appsub env e1)
                       (appsub env e2)
                       env
            )))

(de concilier (t1 t2 env)
    (cond
     ; Les deux termes sont déja égaux.
     ; Aucune modification à la pile.
     ((equal t1 t2) env)
     ; L'un des termes est une variable.
     ; Test d'occurrence et ajout à la pile d'une substitution.
     ((var? t1 (ifn (occ t1 t2)
                    (cons (cons t1 t2) env)
                    (impossible
                     (list 'non 'occur_check t1 'dans t2))))
     ((var? t2 (ifn (occ t2 t1)
                    (cons (cons t2 t1) env)
                    (impossible
                     (list 'non 'occur_check t2 'dans t1))))
     ; L'un des termes est une constante et l'autre n'est pas une
     ; variable. L'unification n'est pas possible. (Etape 4 de
     ; l'algorithme de Robinson).
     ((or (atom t1) (atom t2))
      (impossible (list 'non t1 'et t2 'non 'unifiables)))
     ; Les deux termes sont composés : divide ut regnes !
     (t (let ((nenv (concilier (car t1) (car t2) env)))
```

```
                (concilier (appsub nenv (cdr t1))
                           (appsub nenv (cdr t2))
                           nenv)
        ))))
```

La fonction "occ" effectuera le test d'occurrence :

```
(de occ (v exp)
    (cond
      ((null exp) nil)
      ((or (var? exp) (atom exp)) (eq v exp))
      (t (or (self (car exp)) (self (cdr exp))))
      )))
```

## 2.3. Le nouvel interpréteur.

Les seules fonctions à modifier sont "résoudre" et "démontrer". Cependant, nous introduirons une fonction renommant les variables apparaissant dans une clause avant son utilisation. Celle-ci est nécessaire car nous n'avons pas implémenté, pour des raisons de simplicité, le mécanisme des chronologies sur la pile d'environnement. La fonction "alpha" créera une nouvelle génération de variables, c'est à dire une nouvelle série d'identificateurs tandis que la fonction "alphaconv" les substituera aux anciennes. Le but est naturellement de limiter la portée de chaque variable à une seule clause.

```
(de alpha (exp)
    (mapc *lvar*
          (lambda (*v*)
                  (let ((newvar (gensym)))
                       (put newvar 'VAR 't)
                       (set *v* newvar)
                  )))
    (alphaconv exp))

(de alphaconv (exp)
    (cond
      ((var? exp) (eval exp))
      ((atom exp) exp
      (t (cons (alphaconv (car exp)) (alphaconv (cdr exp))))
    ))
```

Nous pouvons dès lors réécrire les fonctions centrales du moteur d'inférences :

```
(de résoudre (résolvante env)
    (if (null résolvante)
        ; La réponse du système est l'environnement crée.
        env
        ; Avancer sur les buts en tenant compte de l'environ-
        ; nement crée jusqu'alors et avec une sortie
```

```
        ; immédiate en cas d'échec.
        (escape echec
                (démontrer résolvante
                           (alpha (get (caar résolvante)
                                       'règles))
                           env)
        )))

(de démontrer (liste_buts liste_axiomes env)
    (if (null liste_axiomes)
        (échec (list 'no_match))
        (let
         ((uni (unifier (car liste_buts)
                        (caar liste_axiomes) env)))
          (cond
            ; Echec de l'unification.
            ((eq (car uni) 'non)
             (démontrer liste_buts (cdr liste_axiomes) env))
            ; L'unification a réussi. Pas de reprise possible.
            ((null (cdr liste_axiomes))
             (résoudre (append (cdar liste_axiomes)
                               (cdr liste_buts))
                       uni))
            ; L'unification a réussi. Possibilité de reprise.
            (t
             (let ((essai (résoudre (append (cdar liste_axiomes)
                                            (cdr liste_buts))
                                    uni)))
                  (if (eq (car essai) 'no_match)
                      ; Retour arrière.
                      (démontrer liste_buts (cdr liste_axiomes)
                                 env)
                      ; L'essai est transformé !
                      essai)))
          )))))
```

Et naturellement, nous adapterons la réponse au problème : lorsque la question sera démontrée, nous fournirons cette fois à l'utilisateur la substitution qui l'établit.

```
(de traiter (liste_buts)
    (let ((réponse (résoudre liste-buts)))
         (if (eq (car réponse) 'no_match)
             (print '(Non. Echec.))
             (écrivar liste_buts réponse))
    ))

(de écrivar (texte env)
    (let ((lv (lisvar texte)))
         (if (null lv)
             (print '(Oui. OK.))
             (mapc lv
                   (lambda (v)
                           (print (list v '= (appsub env v))))
             ))))
```

```
(de lisvar (texte lv)
    (cond
      ((var? texte) (if (member texte lv) lv (cons texte lv)))
      ((atom texte) lv)
      (t (lisvar (car texte) (lisvar (cdr texte) lv)))
    ))
```

Le système ainsi construit ne nous assure pas encore le non-déterminisme. En effet, celui-ci s'arrête obstinément à la première réponse satisfaisante. Nous pouvons remédier à cet inconvénient en perfectionnant la fonction démontrer. Il suffit de forcer le retour arrière après chaque réponse.

```
(de démontrer (liste_buts liste_axiomes env)
    (if (null liste_axiomes)
        (échec (list 'no_match))
        (let
         ((uni (unifier (car liste_buts)
                        (caar liste_axiomes) env)))
          (cond
            ((eq (car uni) 'non)
             (démontrer liste_buts (cdr liste_axiomes) env))
            ((null (cdr liste_axiomes))
             (résoudre (append (cdar liste_axiomes)
                               (cdr liste_buts))
                       uni))
            (t
             (let ((essai (résoudre (append (cdar liste_axiomes)
                                            (cdr liste_buts))
                                    uni)))
                  (if (eq (car essai) 'no_match)
                      ; Retour arrière.
                      (démontrer liste_buts (cdr liste_axiomes)
                                 env)
                      ; L'essai est transformé !
                      (and (écrivar lu essai)
                           (print 'ou 'encore)
                           (démontrer liste_buts
                                      (cdr liste_axiomes) env)))))
          ))))
```

## 3. Le prédicat évaluable "cut".

Dans cette dernière phase, nous allons donner à l'utilisateur la possibilité de disposer du prédicat évaluable "cut" qui permet, comme nous le savons d'élaguer des branches entières de l'arbre de recherche.

Les modifications à apporter à notre précédent interpréteur se limiteront au fonction "démontrer" et "traiter".

```
(de dèmontrer (liste_buts liste_axiomes env)
    (cond
     ; Le prochain but est un "cut".
     ((eq (car liste_buts '/)
      (let ((essai (rèsoudre (cdr liste_buts) env)))
           (if (memq (car essai) '(no_match cut))
               (slash ratè)
               essai)))
      ((null liste_axiomes) (èchec (list 'no_match)))
      (t (let
          ((uni (unifier (car liste_buts) (caar liste_axiomes) env)))
          (cond
            ((eq (car uni) 'non)
             (dèmontrer liste_buts (cdr liste_axiomes) env))
            ((null (cdr liste_axiomes))
             (rèsoudre (append (cdar liste_axiomes) (cdr liste_buts))
                       uni))
            (t
             (let ((essai
                    (escape slash
                            (rèsoudre (append (cdar liste_axiomes)
                                              (cdr liste_buts))
                                      uni))))
                    (cond
                      ((memq (car essai) '(no_match cut))
                       (dèmontrer liste_buts (cdr liste_axiomes) env))
                      ((eq essai 'ratè) (èchec 'cut)))
                      (t (and (ècrivar lu essai)
                              (print 'ou 'encore)
                              (dèmontrer liste_buts
                                         (cdr liste_axiomes) env))))
        )))))))

(de traiter (liste_buts)
    (let ((rèponse (rèsoudre liste-buts nil)))
         (if (memq (car rèponse) '(no_match cut))
             (print '(Non. Echec.))
             (ècrivar liste_buts rèponse))
    ))
```